Cromemco
16K
Extended
BASIC
Instruction Manual

Cromemco
16K
Extended
BASIC

Cromemco
incorporated
Tomorrow's Computers Today
280 BERNARDO AVE. MOUNTAIN VIEW, CA 94043

TABLE OF CONTENTS

INTRODUCTION

1.0   Design Criteria of Cromemco 16K Extended BASIC

Many different versions of BASIC are commercially
available from computer manufacturers. These
different versions vary greatly in computational
precision, programming power, ease of programming,
speed of execution, and amount of memory required
for the BASIC program. There are a number of
BASICs available which are similar in capability to
the original BASIC developed at Dartmouth College.
These BASICs typically require about 8K of memory.
There are also a number of tiny BASICs available
that require only 2K of memory but which have
different or less capability than an 8K BASIC. For
many of today's demanding applications, more
features are required than were provided in the
original Dartmouth BASIC. A BASIC that provides
advanced features and capabilities is frequently
called an Extended BASIC. The features included in
Extended BASICs also vary widely among computer
manufacturers.

Cromemco 16K Extended BASIC (which was developed
exclusively for Cromemco by Shepardson
Microsystems, Inc.) is designed to maximize
computational precision, programming power, and
speed of execution by fully utilizing the extensive
158 instruction set of the Z-80 microprocessor.

Cromemco's powerful Extended BASIC was specifically
designed to meet the most demanding requirements of
business firms (such as inventory and accounting
programs) while providing the flexibility and speed
necessary for real-time control applications in
both industry and the home. A brief review of some
of the features included in Cromemco 16K Extended
BASIC follows.

One major feature of Cromemco 16K Extended BASIC is
rapid, 14-digit arithmetic using the powerful,
binary-coded decimal (BCD) arithmetic instructions
which are unique to the Z-80 microprocessor. All

arithmetic operations are performed using the 16 high-speed registers of the central processor for intermediate storage. The speed and precision obtained when performing BCD arithmetic cannot be obtained on 7-register microprocessors such as the 8080. Many BASICs were designed for six or seven digit precision using computer-oriented binary arithmetic. It is possible to do calculations which are as accurate using binary arithmetic as they are using BCD arithmetic. However, many BASICs which use binary arithmetic have introduced highly visible errors when calculating dollar and cent conversions in business and financial programs. These conversion errors cannot occur in BASICs which utilize BCD arithmetic. In addition to arithmetic operations, all functions in Cromemco 16K Extended BASIC are computed to 14-digit accuracy. Many other BASICs that provide 14-digit accuracy do so at greatly reduced execution speeds and often without carrying full accuracy through all subroutines.

Real-time control applications often require integer, 16-bit (4 1/2 digit) arithmetic. Cromemco 16K Extended BASIC provides this capability along with direct memory, input/output, and machine-code subroutine access through BASIC instructions. There is also often a need in real-time control system applications to store large arrays of data. Cromemco 16K Extended BASIC offers the flexibility of allowing the user to manipulate, store, and retrieve six-digit floating point numbers. This abbreviated form of the highly efficient, 14-digit mathematics capability is also slightly faster in execution than the 14-digit format.

Furthermore, in Cromemco 16K BASIC (unlike most microcomputer BASICs), constants used in program lines cause execution speeds as fast or faster than variable references. In addition, integer constants occupy no more room than variable references.

Execution speed is maximized in 16K Extended BASIC by the utilization of a semi-compiling design. The semi-compiling design provides many of the important features of both a compiler and an interpreter. In other words, the properties of a compiler which provide high execution speed are combined with the interactive capabilities and programming ease provided through an interpreter.

For example, programs may be stopped during
execution, examined, and modified. The computer
may then be instructed to continue to run the
program without affecting the proper execution of
the program. The ability to stop, examine, and
modify programs can greatly reduce the time and
cost associated with computer programming. This
interactive capability is not available with
optimized compilers. The semi-compiling operation
of 16K Extended BASIC is performed after each
program line is entered. Any syntax errors are
detected at the time of entry rather than during
execution. Consequently, the time required to
enter and debug a program is reduced substantially.

Some versions of BASIC have either rudimentary or
no error message capability. With these versions,
if a fatal error causes the program to crash or
halt unexpectedly during execution, it is up to the
user to reconstruct the failure within the program.
Other versions will produce a numerical value after
the program fails. This value must be interpreted
from a list of error messages. Some more powerful
BASIC programs will return an English-language
statement explaining the failure after the fact.
Cromemco 16K Extended BASIC not only generates
English-language error messages, but it also
examines each statement as it is entered and prints
error messages immediately. The value of such pre-
checking, using the error messages, can be
pronounced. This capability is extremely important
when making corrections to real-time control
systems where inadvertent errors in modifications
can be costly. In general programming, the
interactive aspect of the error messages is
extremely convenient and can substantially reduce
programming time and errors.

Dynamic error trapping is provided in Cromemco 16K
Extended BASIC for specialized routing of program
control in response to BASIC generated error
conditions. Using the error trapping feature,
overflow and underflow arithmetic errors can be
handled without operator assistance. The value of
this feature in preparing programs that will be
used by unskilled operators cannot be
underestimated. For instance, should a disk
read/write error occur, 16K Extended BASIC can be
programmed to type out a message such as: CALL MR.
HIGGINS (415-030-1234) IN THE ACCOUNTING DEPARTMENT
thereby alerting skilled personnel who can correct

the problem.

A TRACE feature is provided in Cromemco 16K BASIC to help determine the origin of errors which are difficult to find in a program being written. Line numbers are logged as they are executed with this feature. FOR loops are automatically indented as they are logged in the TRACE mode.

Cromemco 16K Extended BASIC also has advanced floppy disk input/output capabilities. For example, both programs and data may be stored on a disk. Programs may be stored in internal machine (binary) or ASCII format. Numeric and ASCII string data may be stored on a disk using sequential or random access methods. The number of files which may be opened simultaneously is limited only by the amount of memory the user allocates for file purposes, and an unlimited number of files may be opened and closed during the course of a program. Chaining of program segments too large to fit in memory at one time can be performed with parameters passed through disk files or memory. File names can be string variables and can be changed dynamically (at runtime) by the BASIC program. BASIC programs can be written, modified, or run by other BASIC programs. Line numbers can be altered and two programs can be retrieved from the disk and concatenated into one program. The versatility provided through the interaction of the Cromemco Disk Operating System (CDOS) and the disk input/output capabilities of Cromemco 16K Extended BASIC is uncommon.

Many BASICs have string handling capability. However, many of these limit the number of characters that can be stored in the string to a maximum of 255. In Cromemco 16K Extended BASIC, up to 32,767 characters or the available memory space are allowed -- whichever limit is smaller. Substrings are utilized to rapidly manipulate these strings. Again, the design of the language emphasizes speed, versatility, and accuracy. A special, high-speed, string-searching function is provided to increase throughput in string manipulation and sorting operations (such as are commonly found in mailing list processing programs).

Program listings from 16K BASIC are print justified and are uniformly spaced for reading ease and

clarity of documentation. FOR loops are also
automatically nested and indented by the listing
function. In the semi-compiling operation, the
only statements that are directly stored in memory
at entry time are the REM comment statements (and
even here spaces are compressed). All other
statements are semi-compiled and stored in machine
readable format. When listed, this machine
readable format is reconverted into ASCII format
and then printed. Thus all spacing and other
variations are lost in the conversion process.
This conversion process allows Cromemco 16K BASIC
to efficiently store large programs by storing
programs only once in semi-compiled format. In
contrast, many other BASICs always maintain an
uncompiled, one-to-one representation of what was
typed in, along with any internal, execution
related storage. Such BASICs require extra program
storage if blank spaces are typed inadvertently
into the program text. Blank spaces can slow down
program execution. Cromemco 16K Extended BASIC
does not store, nor is it affected by,
inadvertently typed blank spaces. Furthermore,
these extra blank spaces are not listed in the
program output.

There are a number of features commonly found in
most Extended BASICs that are included in Cromemco
16K Extended BASIC. These features include multi-
statement lines; one, two, and three dimensional
arrays; 26 user-definable functions; advanced
formatting capabilities with PRINT USING, TAB,
SPACE, and SYS/SET functions; and direct machine
language interaction with INP, OUT, PEEK, POKE, and
USR instructions. Finally, Cromemco 16K Extended
BASIC also has flexible INPUT and PRINT routines
which are capable of supporting a wide range of
terminals and line printers.

This manual was designed for use both by users who
have no prior experience with BASIC and by
experienced programmers. Users with little or no
prior experience are encouraged to read sections
1.0 through 1.3 and then proceed to Chapter 7,
Programming Examples. These examples are designed
to introduce Cromemco 16K Extended BASIC to the
novice programmer. Chapter 19 is a Glossary and
may be consulted if the definition of a term is in
question.

Both the beginner and experienced programmer will

gain insight into BASIC's file structures and
access methods by reading Chapter 15, Discussion of
Data File I/O.

The advanced user will want to refer to Chapters
17, 18, and 21 (the Appendix) as well as to section
1.4, A Technical Discussion.

1.1   Computers and Computing Power

A computer is a device which performs high-speed
mathematical or logical calculations or which
processes information derived from coded data in
accordance with a predetermined program.   This
predetermined program is a set of instructions
arranged in logical sequence which directs the
computer to perform a desired operation.

All computers are based on one fundamental concept
-- the binary digit. The binary number system uses
only the digits 0 and 1.   The binary digit concept
is so useful because it can be represented by any
device which is capable of assuming one of two
possible stable states (on and off).   A computer,
even in its most complex form, is simply a
collection of logic gates which respond to the
presence or absence of a voltage. The presence or
absence of such voltages, called logic levels,
makes it possible to represent binary conditions in
a logic circuit. Typically, these logic levels are
defined as 0's for off states and 1's for on
states.

Binary digits, commonly called bits, can be used to
represent numbers of any magnitude by grouping
digits together. The primary level at which binary
digits are grouped within a particular computer is
called word size. Most microcomputers use an 8-bit
word size. This 8-bit unit is called a byte. If
we consider the various computer words we can make
out of all possible combinations of eight bits, we
find that there are 2 to the 8th power or 256
possible combinations. We can then build a
computer which will respond differently to each
byte which we present as an instruction.   For
example, an instruction of all 0's might mean stop,
while 10101010 might mean increment a number by 1.
Furthermore, a particular byte might stand for one
operation when used with one model of computer and

might stand for a very different operation in
another computer system. The important point is
that all computers execute instructions through
this binary language which is a combination of on-
off states represented by Ø's and l's.

1.2   High and Low Level Languages

As stated in the introduction, a computer must be
programmed to perform calculations or to process
information. Programs are prepared in a language
with precise rules of grammar. Currently available
computer languages fall into one of two major
categories: high-level or low-level languages.
Internally, computers perform various operations
according to the binary language described earlier.
This binary language is commonly called machine
language. Machine language is at the lowest level
of computer languages and consists .of groups of on
and off states. A machine language word is any
combination of eight Ø's and l's, (e.g., Ø11ØØ1ØØ).
As you can see, it is nearly impossible to tell
what a given machine language word means by simply
looking at the binary word. Furthermore,
programming a computer in this fashion would be
both time-consuming and tedious.

To simplify the programming process, some easier
method of entering and recognizing instructions is
required. The next step in low-level languages
then, is an assembly language which uses short
mnemonic symbols that assist in the definition of
the instruction. An assembly language translates
simple instructions, such as HALT or INCREMENT,
into the appropriate machine language instructions.
But even this form of programming is too cumbersome
for many applications. In assembly language, a
program to print out "HELLO" on a terminal may
require as many as 5Ø steps. Furthermore,
individuals without a great deal of computer
expertise often find it difficult to translate
problems into a computer program at either the
assembly or machine language level.

As a result, higher level languages were developed
with instructions and rules of grammar which are
easier to learn and apply. Problem-oriented, high
level languages include FORTRAN, COBOL, PASCAL, PL-
-1, and a variety of other languages. These high-

level languages are translated into machine language through programs called compilers. Any language that must be compiled or translated one or more times before it is reduced to the machine code is said to be a high-level language.

1.3 Basic and Its Uses

Most of the high-level languages mentioned in the previous section were designed for use by special groups comprised of relatively sophisticated computer users. Consequently, most of these languages are somewhat difficult to learn. However, as the computer moved from the design laboratory into the university laboratory and then on into business, home, and school usage, the need for an easy to learn, grammatically simple, English language based, programming language grew. To answer this need, Professors John G. Kemeny and Thomas E. Kurtz of Dartmouth College developed the first version of BASIC in 1965. BASIC is a higher level language which is easy to learn and which can be applied to most computing problems.

In addition to being easy to learn, BASIC has a number of features which make it useful for two very important and common applications: time-sharing systems and interactive programming. In BASIC, simple, English language instructions are translated line by line as the program executes into machine code through a program called an interpreter. This interpreter makes it easy to interrupt the conversion process, proceed to another task, and then return to pick up where you left off. Consequently, a computer can be used to allow multiple users running BASIC to timeshare a system. In timesharing, the computer handles each user in some specified sequence. The high speed of the computer, however, makes it appear that all users are being handled simultaneously.

BASIC is also particularly well-adapted for use in interactive programs which require continual user interaction with the computer. Interactive programming allows the user to continually update and revise a program based on intermediate results. For complex problems involving a number of parameters that may require changing before a solution is reached, the interactive programming

capability is a powerful tool.

Because of these important features, the library of
programs written in BASIC, many of them in the
public domain, has expanded rapidly. Many complex
business programs, including inventory control,
payroll, shipping and receiving, and accounting,
are available in BASIC. There are programs written
for computer assisted instruction (CAI) and for a
wide range of interactive games in which users
compete with each other or with the computer in
simulations of a variety of situations and events.

In addition, the same computer system may be used
to streamline accounting procedures, control
manufacturing processes, regulate energy usage, or
play games. Only the BASIC program need be
changed. With the introduction of inexpensive,
microcomputer-based, time-sharing systems (such as
the multi-user version of the Cromemco System Three
Computer), true multi-user computer operation
becomes available at only a fraction of its cost
five years ago.

1.4   A Technical Discussion of the
        Cromemco 16K Extended BASIC Language

A sizable portion of the material in this section
will be of interest to those who might be relative
novices in the field of system-level programming.
Also, it is hoped that more advanced programmers
will appreciate the information presented relating
to the internals of Cromemco 16K BASIC.

The 16K Extended BASIC Language is an incremental
compiler written in Z-80 code by Shepardson
Microsystems, Inc. of Cupertino, CA specifically
for, and to the specifications of, Cromemco, Inc.

An incremental compiler is a type of interpreter
which not only fully checks for correct syntax as
the source is being entered, it also immediately
resolves any references to undefined items. The
primary advantage of the incremental compiler is
execution speed. Since variable and line number
references are already resolved, no execution time
is spent searching tables for items such as a
variable name match.

1.4.1   Syntax Related Tables

As each statement line (or direct command instruction) is entered from the keyboard (or some other I/O device), it is checked for correct language syntax and converted into an internal format. Actually, the routine which does the syntax checking also converts source items into internal tokens at the same time. The BASIC syntaxer is actually <u>driven</u> by a set of syntax tables, the purposes of <u>which</u> are described below.

The <u>Reserved Name Table</u> contains a list of all instruction, function, and operator names that are recognized by BASIC. This table is used to equate the user's ASCII input to a particular internal token. Thus in addition to the reserved name, each entry contains the token to be used.

<u>Main Syntax Table</u>. Once the first reserved name in an instruction has been found, the instruction may be classified as to type. The type directs BASIC to an entry in the main syntax table, where an entry is made up of a list of required and/or optional further items (actually their equated tokens) that must be found in the statement. If a required item is not found in the proper position in the input line, a syntax error is generated. This table is quite complex, as even such things as arithmetic expressions in all their variations must be covered. There are even syntax subroutines included here, so that a statement type may require (for example) a subscripted string name by calling for a subroutine which in turn requires a string variable, a left parenthesis, subscripts, and a right parenthesis. (Please note that this is not actually how the search is performed; it is actually more segmented and more highly organized than stated here.)

The syntax table contains other information about instruction types, such as whether they are allowed in direct mode (a command), within a program (a statement), or both places. It can also declare that an instruction must be the last one in a line or that subsequent instructions on the same line are permitted.

The <u>Binding Strength Table</u> is more properly

described as a run-time table, since it determines
the priority in which operators (+, -, AND, NOT,
etc.) are to be executed. It is included here
because it is used to relate the internal tokens
for the various operators to their priorities.

1.4.2 On Listing a Program

After a program has been converted to internal
token form, it is obviously desirable to be able to
implement the LIST instruction. On unaltered
source types of interpreter, this is simply a
matter of dumping the memory contents back out to
the list device. In Cromemco BASIC, the problem is
significantly more complex. Each token must be
reconverted to its ASCII form and (as we shall see
later) constants must be reconverted to external
format.

1.4.3 Tables in User Program Memory Space

Generally, these types of tables are referred to as
runtime tables, though many of them are established
at program entry time. They are the fundamental
tables which the interpereter uses in executing a
user program.

The Statement Table is not strictly speaking a
table. It is simply an area of memory where all
program lines are stored after they have been
converted to internal format. The most
distinguishing feature of this table in Cromemco
BASIC is that the last line entered is always at
the end of the table. Only when a line is deleted
or modified is it necessary to slide the table (in
order to recover the space occupied by the deleted
parts). Also note that the program's line numbers
are not stored in the statement table. Instead,
the line numbers are found in the line number
table.

Line Number Table. This table is also known as the
Label Table. Herein are stored the actual line
numbers. Also in this table each entry has the
address of the beginning of its corresponding
program line within the statement table. The third
item in each entry in this table is the address

within the table of the next logical succeeding
line (i.e., the line next to be executed unless
program flow is altered by a GOTO, GOSUB, or NEXT
statement). Note the implication here: even this
table is not organized in line number order! As
each new line is entered, its label entry is simply
appended to the table and the logical successor
pointers are updated as needed.

Variable Table. When a variable is first defined
(which occurs as the first line using it is
entered), an entry is created for it in this table.
There are many types of information contained
herein. All variables have their name ('Al', 'A$',
'E') and variable type (string, integer, short or
long floating point, and/or array) stored here. In
addition, scalar variables (non-array numeric
variables) have their actual value stored here.
Strings have their dimension and current length
stored also. Arrays have the number of dimensions
in use noted as well as the maximum value of each
dimension. In addition, arrays and string entries
contain the address of the actual location of -the
data for each. The data so addressed is located
within the string/array table.

String/Array Table. Again, this table is not so
much a table as simply a section of memory reserved
for the storage of array and string data. Of the
tables mentioned so far, this one is unique in that
it is allocated (expanded) at runtime. That is,
since statements of the form:

    310 INPUT J
    320 DIM A$(J), B3(J,2)

are legal, arrays and strings cannot have their
space allocated to them until they are actually
dimensioned during program execution.

The For/Next Table. An entry is placed in this
table whenever a FOR instruction is encountered.
Information kept here includes the address of the
corresponding variable table entry, the size of
STEP requested (or implied), the terminating TO
value, and the address of the entry in the label
table corresponding to the line logically following
the FOR instruction. If a FOR statement is
executed using a variable with an entry already
stored in the FOR/NEXT table, the old entry is
deleted, the table is adjusted, and the new entry

is appended.

GOSUB/RETURN Table. A very simple table, this consists simply of the address of an entry in the label table which corresponds to the line logically following the GOSUB statement.

User-Defined Functions Table. Since only user-defined functions FNA through FNZ are allowed in 16K BASIC this is a fixed length table with 26 entries, each active one of which contains the address of the label entry corresponding to the line where the function was defined.

1.4.4   An Example

There follows an actual example of what happens internally to BASIC when a program line is entered (syntax time) and then executed (run time). It is assumed that any existing program and/or data has been removed via a SCRatch command and that the example line shown is the first (and only, though this is not significant) line entered towards creating a new program.

This is the example line:

        100 IF A2 = 3.7 THEN 300

1.4.4.1 Syntax Time

The first thing detected is the line number, '100'. This line number is placed in the label table along with the address of the beginning of the statement table (where the encoded line will be stored). Since this is the only entry in the label table, the successor pointer will be set to zero to indicate no successor. Since the label table works from the top down, the pointer to the bottom of the table would be updated to indicate where to place the next entry. So far, then, the label table looks something like this:

        2 bytes -- pointer to statement in statement
                   table

2 bytes -- pointer to successor label (zero, currently)

3 bytes -- the line number, 100, in BCD (5 digits significant)

Next, the keyword 'IF' is encountered and recognized as a valid statement type. It is translated to a single byte internal token and the syntax table entry for 'IF' is accessed. The syntax for 'IF' requires that the keyword be followed by an expression, so the appropriate syntax subroutine is called.

Without going into detail as to what a valid expression is, and how the syntaxer determines that it has or has not encountered one, it is sufficient to note that certainly 'A2=3.7' would be considered valid. That being so, the variable 'A2' makes its first appearance and must be added to the variable table. Its entry would look something like this:

2 bytes -- the variable name (A2)

1 byte -- the variable type, assume a 02, short floating point

4 bytes -- the actual value of the variable, in the length required for its type. This is set to zero until altered.

The statement table would receive a byte signifying that a numeric variable had been found followed by two bytes which are the address of the entry for 'A2' within the variable table.

The equals sign ('=') would be recognized as an operator, and its internal code stored next in the statement table.

The value '3.7' would be recognized as a floating-point constant (integer constants don't have decimal points), and a routine would be called to translate it to internal format. The statement table would receive a byte signifying a particular type of constant (short-floating here) followed, and then the internal constant would be stored.

The 'IF' syntax requires that the expression be followed by a 'THEN' keyword. When it is found,

its internal token is stored in the statement
table.

'THEN' may be followed by either another whole new
instruction or (as in this case) by a line number.
The processing of this line number is partly what
qualifies the Cromemco BASIC interpreter to be
called an incremental compiler.

The label table is scanned for a line number '300'.
When it is not found, an entry is made for it
similar to the one previously made for line '100'.
However, for line number 300, the address of its
corresponding line in the statement table is set to
an illegal value to indicate that it doesn't exist.
Later, if a line 300 were entered, the address
would be changed to reflect the fact. Also, the
entry for line 100 is updated to reflect that its
logical successor is line 300 (the next
sequentially numbered line).

The statement table receives a token to indicate
that a label table address will follow, and then
the address of the label table entry for line '300'
is stored. Finally, a byte of zero (00) is stored
to indicate the end of line.

At this point, then, the statement table will
contain the following:

      1 byte -- token for 'IF'

      1 byte -- token, a variable table address
                 follows

      2 bytes -- the variable table address for 'A2'

      1 byte -- token for '='

      1 byte -- contains 02 hex to indicate that a
                 short floating point constant
                 follows

      4 bytes -- the internal form of the constant
                 '3.7'

      1 byte -- token for 'THEN'

      1 byte -- token, a label table address
                 follows

        2 bytes -- label table address for line number
                300

        1 byte -- value 00 hex, indicates end of line

1.4.4.2 Run Time

Assume now that the user requested a RUN of the program (and, incidently, even this simple statement must go through the syntax process before it is executed.)

BASIC searches out the first logical line of the program from the label table and then performs the following steps:

First, the address of the statement within the statement table is found. Then the first byte of the statement is used to vector to the routine responsible for processing IF statements.

The 'IF' processor knows that what follows is an expression, so it calls the expression execution routine to evaluate 'A2 = 3.7'.

The expression evaluator notes that the first item is a variable, so it uses the variable address to extract the value of the variable from the variable table entry for 'A2'. It places this value in an area of memory referred to as the argument stack.

The '=' relational operator (note: this is not the same thing as the assignment operator '=' used in LET instructions) is then pushed onto another BASIC-maintained stack known as the operator stack.

The constant token causes the short floating point constant '3.7' to also be placed on the argument stack.

Since there is no more to the expression, the expression evaluator calls on the processing routine for the equal relational operator. The equality processing routine extracts the two values (contents of 'A2' and the value '3.7') and returns either a one (1) if they are equal or a zero (0) if they are not. The expression evaluator is finished and returns with this value to the 'IF' processor.

At this point let us examine what happens if the
value returned is 'l' (which it can't be in our
example). The 'IF' processor ignores the 'THEN'
token (it has to be there, the syntaxer passed it!)
and encounters the label table token. This implies
a change in program flow to the line specified in
the table entry. So the 'IF' processor simply
places the entry encountered in BASIC's own 'next
logical statement' location. It then checks to
ensure that a line really exists to correspond to
the entry. The line does not exist and BASIC
issues the error message, GOTO UNDEFINED LINE
NUMBER.

Now backtrack a bit and assume that the expression
evaluator had returned with a '0' instead (which it
would in our example). In this case, the 'IF'
processor simply aborts processing of its line and
returns to allow BASIC to execute the next logical
line. When BASIC encounters the next logical line
('300' in our example) it checks to see if the line
exists (it doesn't here) and executes it if it
does. If it does not, BASIC ignores the line and
falls through to the next logical line. Since line
300 does not exist, and since there is no logical
successor, BASIC would print out ***END***.

In Cromemco BASIC, there is no speed penalty for
using any and as many variables wherever desired in
the program, there is no reason to put certain
statements (i.e., subroutines) first in the
program, and constants in programs execute as fast
or faster than variables.

INSTRUCTION SYNTAX

The Cromemco 16K BASIC language is designed to allow the user to structure and format a program in a wide variety of styles. This section covers the features which allow this flexibility as well as those elements of BASIC syntax which do affect program operation.

2.1 Spaces or Blank Characters

All spaces are optional within 16K BASIC except for those which are included as part of text (i.e., REMarks, string literals and string variables, see sections 4.3 and 5.2). Once a program has been entered in BASIC, the spacing within a listed program will automatically conform to BASIC standards. The following user interaction with BASIC demonstrates both that spaces are ignored on input, and that BASIC stores a program in a standard format.

```
>>1    ØØPR      INT"This is a string literal"
>>12ØREM The spacing within the string literal
>>13ØREM and the REMark statements will not be
>>14ØREM changed by BASIC.
>>15ØFORI=1TO1Ø
>>16ØPRINT I      ;
>>17ØNE    XT I
>>1   8ØEN      D
>>L IST


100  PRINT"This is a string literal"
120  REM The spacing within the string literal
130  REM and the REMark statements will not be
140  REM changed by BASIC.
150   FOR I=1 TO 10
160   PRINT I;
170   NEXT I
180  END
```

2.2 Upper Case Characters

All letters used in Cromemco 16K BASIC must be upper case (capital). The only exceptions to this are string literals, string variables, and REMarks which may contain any printable characters. If any BASIC instruction is entered in lower case letters, a syntax error will result.

2.3 The BASIC Prompt (>>)

When BASIC is ready to accept a command or statement line, it displays a prompt which consists of two greater than symbols (>>). The purpose of the prompt is to indicate that BASIC has finished its last task and is waiting for the user's next instruction.

2.4 Commands

A command is a BASIC instruction which is executed immediately (as soon as the carriage return is depressed). Commands have no line numbers because they are not stored by BASIC.

Cromemco 16K BASIC allows most instructions to be used as commands. For example, BASIC can be used as a calculator while it is in the command mode:

```
>>PRINT 20000/5
4000

>>PRINT (4000+77)/63.
64.714285714285
```

As can be seen, a command may be given whenever the BASIC prompt (>>) is displayed.

In this manual, the term instruction includes both commands and statements. All instructions listed as instructions or commands may be used as commands, while those listed as statements may not be used as commands.

2.5 Statements

A statement is a BASIC program line which contains one or more instructions, and which is stored for execution at a later time. Statements are not executed until the RUN command, or some other command which will begin execution of a program, is given. A statement line has a unique (within a program) line number by which it can be accessed. If a second statement is entered with the line number of a line which already exists, the original line will be replaced by the new line.

Cromemco 16K BASIC allows most instructions to be used as statements. For example, one BASIC program can load and run another BASIC program:

        100 RUN "PROG2"

Execution of this statement will cause PROG2 to be loaded into the User Area and execution to begin.

In this manual, the term instruction includes both statements and commands. All instructions listed as instructions or statements may be used as statements, while those listed as commands may not be used as statements.

2.5.1 Multiple Instruction Lines

Cromemco 16K BASIC allows more than one instruction to be associated with a single line number. Each pair of adjacent instructions must be separated by a colon (:). The number of instructions which may appear on a single line is limited only by the length of a line. For example:

        100 INPUT A : IF A<0 THEN GOTO 100
        200 PRINT SQR(A) : PRINT : PRINT
        300 REM LINE 200 WILL SKIP 2 LINES

User defined functions (DEF FNs) and DATA statements must appear as a single instruction on a line. The following instructions may appear as part of a multi-instruction line but they must be the last instruction on the line, i.e., no other instruction may follow on the same line.

```
RUN
REM
GOTO
GOSUB
RETURN
FOR
ON...GOTO (GOSUB)
ENTER
DELETE
```

A colon may not terminate a multiple instruction
line. A colon must be followed by another
instruction.

The IF...THEN instruction is unique when followed
by other instructions on the same line. The reader
is referred to the IF...THEN instruction for a
further discussion.

NUMERIC AND STRING

INTERNAL MACHINE REPRESENTATION

Numeric and string alphanumeric information is
stored by BASIC in different forms. This section
explains these different formats.

3.1   Integer

Integers are whole numbers and in 16K BASIC they
must be within the range +32767 to -32768. When
stored by BASIC, an integer occupies 2 bytes of
memory.

Integer numbers are stored low byte first, high
byte second. If the high bit of the high byte is a
1, then the number is negative; if it is a Ø, then
the number is positive.

A positive number is stored as the binary
representation of the number.

A negative number is stored as the 2's complement
of the number.

Example:

The number 1234 will be represented by the
hexadecimal bytes D2 Ø4 when stored as an Integer.
When the order of the bytes is reversed (Ø4 D2)
this is the binary equivalent of the decimal number
1234.

The number -1234 will be represented by the
hexadecimal bytes FB 2E when stored as an Integer.
When the order of the bytes is reversed (2E FB)
this is the 2's complement of the binary equivalent
of the decimal number 1234. Because the high bit
of the high byte was 1 (before the 2's complement
was taken), the number is negative (-1234).

3.2   Short Floating Point

A Short Floating Point number stored by BASIC
occupies 4 bytes, has an accuracy of 6 digits, and
must be within the range +9.99+62 to +9.99E-65.

The first byte of a Short Floating Point number
contains the sign of the number and the exponent in
excess 40H (64 decimal) notation. If the high bit
of the first byte is a 1, then the number is
negative; if it is a 0, then the number is
positive. Note that this is not the sign of the
exponent but rather of the number itself. The
remaining 7 bits of the first byte contain the
exponent plus 40H. In order to find the true
exponent, 40H (64 decimal) must be subtracted from
this number.

The remaining three bytes contain the BCD (Binary
Coded Decimal) mantissa which has been normalized
to a value between 0 and 1. The implicit decimal
point is located before the first byte of the
mantissa. Each byte of the mantissa can contain 2
significant digits yielding a total of 6
significant digits for a Short Floating Point
number.

Example:

The number -1.2345E+21 will be represented by the
hexadecimal bytes D6 12 34 50 when stored as a
Short Floating Point number.

The first bit of the first byte is a 1 indicating
that the number is negative. The remaining 7 bits
of the first byte (56H or 86 decimal) is the
exponent plus 40H. To get the true exponent 40H
must be subtracted from 56H. This leaves an
exponent of 16H or 22 decimal. This is not the
exponent of the original number because the number
was normalized. In this example, normalization
involved dividing the number by 10 and adding 1 to
the exponent to compensate for the division.

The remaining three bytes are the BCD
representation of the normalized number. BCD
stands for Binary Coded Decimal which is a method
of representing a decimal number in binary. Using
this method, each byte can contain two one digit

32

decimal numbers. As can be seen from the example,
non-significant digits are zero filled.

3.3   Long Floating Point

A Long Floating Point number stored by BASIC
occupies 8 bytes, has an accuracy of 14 digits, and
must be within the range +9.99+62 to +9.99E-65.

The internal representation of a Long Floating
Point number is similar to that of a Short Floating
Point number.   The difference is that four
additional bytes are added to the mantissa for a
total of 7 bytes.   This is how a Long Floating
Point number can maintain 14 significant digits.

3.4   Hexadecimal

A Hexadecimal number occupies two bytes and must be
within the range ØH to FFFFH.

The internal representation of a hexadecimal number
is the same as that of an Integer. Except for the
functions BINAND, BINOR, and BINXOR, hexadecimal
numbers are treated as signed integers (refer to
section 3.1).

3.5   String

A string is an ordered list of alphanumeric
information.   Examples of strings include words,
sentences, parts of words, groups of letters or
special characters.

Elements of a string (characters) stored by BASIC
occupy 1 byte each.   They are represented
internally as an eight bit number which is the
ASCII (American Standard Code for Information
Interchange) code for the character being stored.
Bit number 7 (the high bit) is a parity bit and,
although it effects string comparisons, it does not
effect the character which is PRINTed.

CONSTANT AND STRING LITERAL FORMATS

Constants are, as the name implies, unchanging.
They each have one value.

String literals are similar to constants in that
they each maintain one value which does not change.

This section covers the standard formats for
constants and string literals in 16K BASIC.

4.1 Integer and Floating Point Constants

A constant is a number. It does not change value
and is represented as it would be in any arithmetic
computation.

There are three types of constants: integer,
floating point, and hexadecimal. All constants
(not specified as hexadecimal) equal to or greater
than 10,000 and those containing a decimal point
are always stored by BASIC as floating point
numbers (either Short or Long Floating Point
depending on the current mode, but always Long if
there are more than 6 significant digits). All
constants (not specified as hexadecimal) with a
value less than 10,000 and not containing a decimal
point are stored by BASIC as Integer numbers.

Constants

Floating Point	Integer
20000	55
3.	9985
.000376	5
12.7	458

4.1.1   Storage of Floating Points Constants

Floating point constants (1.2, 3., 1E6, etc.) are stored according to the mode which is active when they are entered. An exception is a constant with more than 6 significant digits, which is always stored as a Long Floating Point number. For example, if the active mode is set to Short Floating Point (see the SF MODE instruction) and the following commands are given:

```
>>LONG L
>>L = 1./3.
```

L will be assigned a value of 0.333333000000000 because 1. and 3. are SHORT (SFMODE was current when they were entered). But, under the same circumstances, if we had said:

```
>>LONG L
>>L = 1./3.00000000
```

L would have been assigned a value of 0.33333333333333 because 3.00000000 is forced to LONG (more than 6 significant digits).

If in the two previous examples, the current mode had been Long Floating Point, L would have received a value of 0.33333333333333 in both cases because both 1. and 3. would have been stored as Long Floating Point numbers.

4.2   Hexadecimal Constants

Hexadecimal numbers are used in base 16 arithmetic. The set of digits used in hexadecimal arithmetic is 0 through 9 and A through F.

In Cromemco 16K BASIC, hexadecimal constants are identified by leading and trailing percent (%) signs.

Constants

Hexadecimal	Equivalent Decimal
%8000%	-32768
%9000%	-28672
%FFFF%	-1
%9%	9
%A%	10
%F%	15
%10%	16
%80%	128
%FF%	255
%100%	256
%7FFF%	32767

A hexadecimal number is stored in the same format as an integer and may be used wherever a constant is allowed.

4.3   String Literals

A string literal is a string which is enclosed between quotation marks. The quotation marks are not part of the string itself, but are used to delimit (mark the ends of) the string. The value of a string literal does not change.

A quotation mark can be represented within a string literal by the use of two quotation marks, one immediately following the other.

String Literals

```
"This is a string literal"
"Here are imbedded ""quotation"" marks"
"{special} *+- characters are o.k.!"
"----------------------------"
"                    just spaces"
```

The trailing quotation mark on a string literal is not required at the end of a line. In this case, the carriage RETURN will terminate, but not be a part of, the string literal.

VARIABLE REPRESENTATION

Variables are, as the name implies, able to change
or vary in value. As a program is executed,
variables may be assigned new values at any time.
This section covers the standard formats for
variables in 16K BASIC.

5.1   Numeric Variables

Numeric variables may be assigned numeric values.
The range and accuracy of these variables depends
on their type. See Chapter 3 on Numeric Internal
Machine Representation for more information.

5.1.1   Format of Numeric Variables

Numeric variables are represented by a single
letter (A-Z) or a single letter followed by a
single number (0-9). There are 286 possible
numeric variable names (A, A0, A1, A2,...,A9, B,
B0, B1,...,B9, C, C0, C1,...,Z, Z0, Z1,...,Z8, Z9).
In compliance with the rules of 16K BASIC, only
upper case letters may be used in variable names.

5.1.2   Integer Variables

Specific variables may be set to Integer by the
INTEGER instruction. The default mode for
variables may be set to Integer if the IMODE
instruction is given before the RUN instruction.
Refer to the sections covering Integer Internal
Machine Representation, and the INTEGER and IMODE
instructions.

5.1.3   Short Floating Point Variables

Specific variables may be set to Short Floating
Point by the SHORT instruction. The default mode
for variables may be set to Short Floating Point if
the SFMODE instruction is given before the RUN
instruction. Refer to the sections covering Short
Floating Point Internal Machine Representation, and
the SHORT and SFMODE instructions.

5.1.4   Long Floating Point Variables

Specific variables may be set to Long Floating
Point by the LONG instruction. Long Floating Point
mode is normally the default mode. If the mode has
been changed to SFMODE or IMODE, the default mode
for variables may be reset to Long Floating Point
if the LFMODE instruction is given before the RUN
instruction. Refer to the sections covering Long
Floating Point Internal Machine Representation, and
the LONG and LFMODE instructions.

5.1.5   Matrices and Lists

A matrix is an array of numeric variables in a
prescribed form. For example, the array:

$$
\begin{array}{rrr}
3 & 2 & 0 \\
1 & 4 & 6 \\
-3 & 4 & 5
\end{array}
$$

is a matrix with three rows and three columns. A
matrix with m rows and n columns is written:

a11 a12 a13 ...a1n

a21 a22 a23 ...a2n
 .
 .
 .
am1 am2 am3 ...amn

The individual entries in the matrix are called
elements or cells. For example the quantity aij in
the above matrix is the element in row i and column

j. Subscripts used to indicate elements always denote the row first and the column second. Cromemco BASIC permits the user to define one, two, or three dimensional matrices. A two (i.e., Mij) or three (i.e., Mijk) dimensional matrix is commonly called a table. A one dimensional matrix, a matrix with n columns but only one row, is commonly called a list. For example, the matrix:

3, -1, 5, -8

is a list (or a matrix) with one row and four columns.

A matrix may be defined to be composed of Long or Short Floating Point or Integer variables.

5.1.5.1 Format

A matrix is named in the same manner as a numeric variable. A specific element of a matrix is accessed by the matrix name followed by 1, 2, or 3 indices enclosed in parentheses. These indices may be numeric expressions, variables, or constants.

Singly dimensioned matricies are implicitly dimensioned as 10. Larger singly dimensioned as well as 2 and 3 dimensional matrices must be explicitly dimensioned by the DIMension, INTEGER, LONG, or SHORT instruction.

An element of a matrix may be used anywhere a variable is allowed.

5.2 String Variables

String variables may be assigned alphanumeric values. This includes all letters (both upper and lower case), numbers, and all printable and non-printable characters. Refer to the section on String Internal Machine Representation for more information.

5.2.1   Format

String variables are represented by a single letter
(A-Z) followed by a dollar sign ($) or a single
letter followed by a single number (0-9) followed
by a dollar sign. In compliance with the rules of
16K BASIC, only upper case letters may be used in
string variable names. Some examples of string
variable names are:

         A$
         A5$
         Q0$
         M$
         M1$

5.2.2   Dimensioning String Variables

There is, for most purposes, no limit to the size
(i.e., number of characters) of a string literal
that may be assigned to a string variable.
However, the default value in 16K BASIC for string
size is 11 or fewer characters. If string values
of more than 11 characters are to be assigned to a
variable, the string variable must be DIMensioned.
The DIM statement is used in BASIC to define the
size of a string variable (see the description of
DIM in Chapter 11).

Example:

      10 DIM A1$(20), B$(30), C4$(40)

In this example, the string variable A1$ is
dimensioned to allow for strings up to 21
characters in length, the variable B$ is
dimensioned to allow up to 31 characters, and C4$
is dimensioned for up to 41 characters. Any string
value assigned to a variable which exceeds the
specified dimension is truncated. Consequently,
the programmer should be sure to dimension string
variables to handle the largest string which will
be input.

Note: DIM A$(20) allows a 21 character string
because string bytes are numbered from 0 through
the specified DIM size. Remember that using the
0th element of strings (and arrays) can save memory
space.

A string of length zero is called a NULL string.

5.2.3   Referencing String Variables

String variables may be referenced (for input,
output, manipulation, or comparison) in their
entirety or by parts (substrings). For example, if
the string variable A$ equals "SUBSTRING EXAMPLE",
substrings of A$ include "SUB", "UBSTR", "G EXA",
and any other part of the string.

Substrings are referenced by subscripting string
variables. The four formats for referencing string
variables are discussed here. In these formats
svar refers to a string variable name (refer to
section 5.2.1) and aexp-1 and aexp-2 are arithmetic
expressions, variables, or constants. If the
string variable has not been explicitly
DIMensioned, it can be considered to have an
implicit DIMension of 10.

5.2.3.1 Format-1:   svar

On input (INPUT or GET), if a string variable is
referenced without subscripts, the entire string is
referenced by implication. This means that all
characters, 0 through the DIMension of the string
are referenced. On output (PRINT or PUT), if a
string variable is referenced without subscripts,
the string is referenced from the first character
(character number 0) through the last non-null
character. This is the same as saying that the
string is referenced from character number 0 for a
length LEN (see the LEN string function).

Summary -

On input, if no subscripts are used, the
entire string variable is referenced as
dimensioned.

On output, the string is referenced from
character 0 through the last non-null
character.

5.2.3.2 <u>Format-2</u>: svar(aexp-1)

If a single subscript is used it must be greater
than zero and less than or equal to the DIMension
of the string variable. This format defines a
substring which starts with the character in
position aexp-1 and includes all subsequent
characters in the string through the DIMensioned
value (input) or through the last non-null
character (output). If, on output, a character
position greater than the LENgth of the string is
explicitly referenced, then all subsequent null
characters are referenced by implication.

If aexp-1 is equal to zero or less, the entire
string will be referenced (character 0 through the
DIMensioned value).

If aexp-1 is greater than the DIMension of the
string variable, a run time error message will
result.

Summary -

If one (aexp-1) subscript is used, the
substring referenced starts with
character in position aexp-1 and includes
all subsequent characters in the string
through the DIMension value (input) or
through the last non-null character
(output).

$0 < \text{aexp-1} \leq$ DIMension of svar

5.2.3.3 <u>Format-3</u>: svar(aexp-1,aexp-2) $\text{aexp-2} \geq 0$

In this format, aexp-1 must be greater than or
equal to zero and less than the DIMension of the
string variable. Thus, aexp-1 defines the starting
character of the substring as it did in format-2.

The second subscript (aexp-2) must be greater than
the first subscript (aexp-1) and less than or equal
to the DIMension of the string variable. Now,
aexp-2 defines the final character of the
substring.

If either aexp-1 or aexp-2 exceeds the DIM value of
the string variable, a run time error will result.

If aexp-1 is less than zero, the entire string is referenced as though no subscripting had been used (see format-1).

If aexp-2 is less than aexp-1, aexp-1 will be the starting character of the substring and the DIMensioned value will be the final character (as though format-2 had been used with aexp-1).

Refer to format-4 if aexp-2 is less than zero.

> Summary -
> If two subscripts (aexp-1, aexp-2) are used, the second subscript being positive, the substring referenced starts with character in position aexp-1 and ends with character in position aexp-2.
>
> $0 \leq$ aexp-1 $<$ aexp-2 $<$ DIMension of svar

5.2.3.4 Format-4: svar(aexp-1,aexp-2)   aexp-2 < 0

In this format, aexp-1 must be greater than zero and less than the DIMension of the string variable. Thus, aexp-1 defines the starting character of the substring as it did in format-2 and format-3.

The second subscript (aexp-2) must be less than zero and its absolute (positive) value must, when added to aexp-1 not exceed the DIMensioned value of the string variable. In this case, the absolute (positive) value of aexp-2 indicates the length or number of characters included in the substring.

If aexp-1 plus the absolute value of aexp-2 is greater than the DIMensioned value of the string variable, a run time error will result.

If aexp-1 is less than zero, the entire string is referenced as though no subscripting had been used (see format-1).

Refer to format-3 if aexp-2 is equal to or greater than zero.

Summary -
If two subscripts (aexp-1, aexp-2) are
used, the second subscript being
negative, the substring referenced starts
with character aexp-1 and continues for a
length of |aexp-2|.

$0 \leq$ aexp-1 < DIMension of svar
aexp-1 + |aexp-2| $\leq$ DIMension of svar.

Note:

If aexp-2 is equal to %8000% then aexp-2 is
considered to be equal to -0 (minus zero) and will
produce a string of length zero (i.e., a null
string).

Examples:

Assume that the command A$ = "ABCDEFGHIJK" has been
given and that the string variable A$ has not been
DIMensioned in a DIMension instruction (it is
implicitly DIMensioned at 10 and it contains 11
characters numbered zero through ten).

Instruction	Result	Explanation
PRINT A$	ABCDEFGHIJK	Format-1 was used to PRINT the entire string.
PRINT A$(5)	FGHIJK	Format-2 was used to PRINT the substring starting with character 5 (remember that the characters are numbered starting with 0).
PRINT A$(4,8)	EFGHI	Format-3 was used to PRINT the substring starting with character 4 and ending with character 8.
PRINT A$(3,-2)	DE	Format-4 was used to PRINT the substring starting with character 3, for a length of two characters.

Now assume the command A$ = "ABCDEFG" has been given. This
will fill the string with the seven characters A-G and 4 null
characters. In the following examples the lower case n
represents a null character.

Instruction	Result	Explanation
PRINT A$	ABCDEFG	Format-1 was used to PRINT character Ø through the first non-null character.
PRINT A$(-1)	ABCDEFGnnnn	Format-2 (with a negative argument) was used to PRINT characters Ø through DIM.
PRINT A$(8)	nnn	Format-1 was used to PRINT characters 8 through 1Ø. Note that character 8 is greater than the LENgth of A$ so that all succeeding nulls are referenced by implication.
PRINT A$(5)	FG	Format-2 was used to PRINT character 5 through the last non-null character.
PRINT A$(5,2)	FGnnnn	Format-3 was used with aexp-2<aexp-1 so the string was output from character 5 through the DIMension of the string.
PRINT A$(8,9)	nn	Format-3 was used to output characters 8 through 9.
PRINT A$(13,15)		Format-3 error, aexp-1>DIM.
PRINT A$(3,%8ØØØ%)		A length of -Ø indicates a null string, no string is output.

OPERATORS

An operator is a symbol or group of letters which indicate that an action is to be taken on one or two items. This section describes the types of operators used in 16K BASIC and the symbols which represent them.

6.1   Arithmetic Operators

The arithmetic operators are analogous to their algebraic counterparts:

Arithmetic Operator	Meaning
+	plus sign (positive number)
−	minus sign (negative number)
** or ^	exponentiation
*	multiplication
/	division
+	addition
−	subtraction

Arithmetic operations in a numeric expression are performed, according to the priority of the operation, from left to right. All operations enclosed within parentheses are performed first. When multiple sets of parentheses appear, the operations in the innermost set of parentheses are performed first, followed by the operations in the next set of parentheses, and so on until the operations in the outermost set of parentheses are evaluated. Following evaluation of expressions enclosed in parentheses, arithmetic operations are performed in the following order: plus and minus signs (unary operators), exponentiation, division and multiplication (these two operations have the same priority), and addition and subtraction (these

operations also have the same priority). When
operations have the same priority, calculations are
performed from left to right within the expression.
The use of parentheses can alter the order in which
operations are performed since the parentheses
override both the left to right priority and the
normal order of operations.

Examples:

        A + B * C ** D

The above expression will be evaluated as follows:

                temp = C ** D
               templ = temp * B
         final value = templ + A

The order of association can be changed by the use
of parentheses:

        ((A + B) * C) ** D

This expression will be evaluated as follows:

                temp = A + B
               templ = temp * C
         final value = templ ** D

It is a good idea to use parentheses if there is
any doubt as to the order in which a series of
operations will be performed. Intermediate results
may also be assigned to temporary variables if this
will help to clarify the order of operations.

6.2   Assignment Operator

The equal sign (=) is the assignment operator. It
is used to assign the value of an arithmetic,
relational, or boolean expression, or a function,
to a numeric variable.

In addition, the assignment operator is used to
assign the value of a string literal, string
variable, or string function to a string variable.

BASIC also uses the equal sign as a relational
operator (see the following section). The only
place the equal sign is a legal assignment operator

is the first equal sign in a LET, implied LET, or
MAT instruction.

Examples:

```
100 A = 2
110 PRINT A
120 A = A*2
130 IF A = 16384 THEN STOP
140 GOTO 110
```

This program will print out all of the powers of 2
which are less than 16384 and then it will stop.
Statement 100 assigns the value of 2 to the numeric
variable A.

Line 120 assigns the result of the multiplication
of the variable A times the constant 2 to the
variable A. This is a good example of the reason
the term assigned to is used instead of equal to.
A is obviously not equal to A times 2. The
expression on the right side of the assignment
operator is evaluated (A * 2) and then this value
is assigned to the variable on the left of the
assignment operator.

Statement 130 uses the equal sign as a relational
operator. The relational expression (A=16384) is
evaluated as true or false. If it is true, the
portion of the instruction following THEN is
executed (program execution STOPs). If it is
false, the next succeeding statement (140) is
executed. Refer to the following section for a
further discussion of relational operators.

Line 140 transfers control back to line 110 and
execution continues until the relational expression
(A = 16384) is true.

```
10 A$ = "END"
20 PRINT "Enter a name,"
30 PRINT "END to stop";
40 INPUT G$
50 IF G$ = A$ THEN STOP
60 PRINT G$ :   PRINT
70 GOTO 20
```

Statement 10 assigns the value of the string
literal "END" to the string variable A$. Line 40
gets a string from the user and assigns the value
of that string to the string variable G$ after

51

having set G$ equal to a string of null characters
(see the INPUT instruction).

Statement 50 uses the equal sign as a relational
operator. The relational expression (G$ = A$ or we
could have used G$ = "END") is evaluated as true or
false and the execution of the program continues on
the basis of the evaluation. See the next section
for more on relational operators.

6.3   Relational Operators

Relational Operator	Meaning
=	is equal to
<	is less than
>	is greater than
<=	is less than or is equal to
>=	is greater than or is equal to
<> or #	is not equal to

Relational operators are used to compare two
expressions. Each of these expressions may be
composed of other relational, boolean, or
arithmetic expressions. This allows the user to
nest relational and boolean expressions.

In addition, relational operators are used to
compare string variables, string literals, and
string functions with each other.

The result of a relational operation is either true
(=1) or false (=0).

Examples:

```
100 IF A<=0 GOTO 150
110 PRINT A=0
120 B=A<0
130 C=A=5
150 END
```

Statement 100 can be read as, "If the value of the

variable A is less than or is equal to zero,
transfer program control to line 150; otherwise
continue with the next statement (line 110)". This
is the most common use of a relational operator.
This statement can be used to test the validity of
user input, transferring control to another section
of code if the input is not as desired.

Statement 110 may at first look like an improper
assignment instruction. Remember that the equal
sign is, in this case, a relational operator, not
an assignment operator. This statement can be read
as, "Compare the value of A with 0 and print the
result of the comparison". Thus, a zero (=false)
will be output if A is not equal to zero and a 1
(=true) will be output if A is equal to zero.

Statement 120 is similar to 110, with the exception
that the result of the relational comparison is
assigned to the variable B. B will therefore take
on a value of 0 or 1.

Statement 130 is a combination of the procedures
used in 110 and 120. Remember that the first equal
sign is an assignment operator while the second
equal sign is a relational operator. This
statement can be read as, "Compare the value of the
variable A with five and assign the result of that
comparison to the variable C". C will take on the
value of 1 (=true) if A is equal to five, otherwise
C will be set equal to 0 (=false).

```
300 S$ = "zzz"
305 INPUT Q$
310 IF Q$ > S$ THEN GOTO 305
315 S$ = Q$
320 PRINT S$
325 GOTO 305
```

Line 300 initializes the string variable S$ by
filling it with z's (the alphabetic character with
the highest value in the ASCII collating sequence).
Line 305 requests string data from the user.

Statement 310 evaluates the relational expression
(Q$>S$) on the basis of the ASCII collating
sequence (refer to the table of ASCII characters in
the appendix). If the first character in the
string Q$ is different from the first character in
the string S$ then these characters are compared
and this relationship determines the relationship

of the two strings. If the first character in each
string is identical to the other, the comparison
moves on the second character and so on. For
example:

Value of Q$	Value of S$	Value of Expression Q$>S$
George	Fred	false
Fred	April	true
April	Ted	false
april	Ted	true
april	ted	false
april	apron	false
april	apricot	true
april	aprilandmay	false

Notice that all lower case letters follow the set
of upper case letters in the ASCII collating
sequence and that if two strings are the same
except for length, the longer string has a greater
value than the shorter one.

Continuing with the example, if the relational
expression (line 310) is evaluated as true, control
is transferred to line 305 and the user is asked
for another string. If the expression is false,
this means that the string which was just INPUT by
the user is closer to the beginning of the ASCII
collating sequence than any other string which had
been INPUT previously. The value of this string is
then assigned to the string variable S$ so that all
strings which are INPUT following this one can be
compared with it. The value of the string is then
PRINTed and control is transferred to line 305.
Execution of this program may be terminated by
depressing the ESCape key.

6.4   Boolean Operators

Boolean
Operator   Meaning

    AND   logical AND

    OR   logical OR

    XOR   logical eXclusive OR

    NOT   logical NOT or negation

Boolean operators perform a logical operation on
one or two expressions. The expressions in a
boolean operation may take on one of two values:
false (=0) or true (=1). In Cromemco 16K Extended
BASIC, all values which are not equal to zero
(false) are considered to be true (=1) when used
with boolean operators.

AND Boolean Operator

The AND boolean operator compares two logical
values and if both are 1 returns a result of 1. If
both values are not 1, then the result is 0.

    Truth Table

AND	0	1
0	0	0
1	0	1

OR Boolean Operator

The OR boolean operator compares two logical values
and if either or both are equal to 1 then the
result is 1. Otherwise the result is 0.

    Truth Table

OR	0	1
0	0	1
1	1	1

XOR Boolean Operator

The XOR boolean operator returns a Ø if the logical
values are identical and a 1 if the logical values
are not identical.

Truth Table

XOR	Ø	1
Ø	Ø	1
1	1	Ø

NOT Boolean Operator

The NOT boolean operator returns the compliment of
any logical value. In other words, if the logical
value is 1, the NOT operator returns a Ø. If the
logical value is Ø, a 1 is returned.

NOT Truth Table

NOT Ø is 1

NOT 1 is Ø

Examples:

```
10 A = 1 : B = Ø : A1 = 1 : B1 = 1
11 IF A AND A THEN PRINT "11 TRUE"
12 IF NOT A THEN PRINT "12 TRUE"
13 IF NOT B THEN PRINT "13 TRUE"
14 IF A XOR A THEN PRINT "14 TRUE"
15 IF NOT (A XOR A) THEN PRINT "15 TRUE"
16 IF A OR B THEN PRINT "16 TRUE"
```

The statements above give some examples of the use
of boolean operators. The IF...THEN instructions
are used in these examples to test if a given
boolean expression is true or false. If the
expression is true, the PRINT instruction following
the IF...THEN is executed. If it is false, then
nothing is printed.

The results of each of these expressions can be
determined from the preceding truth tables. For
example, line 14 uses the XOR operator to compare
two true (=1) values. Looking at the XOR Truth
Table, it can be seen that a 1 and a 1 yield a Ø or
false value. Line 15 negates this same expression.

Looking at the NOT Truth Table, NOT Ø (Not false) is seen to be equal to 1 or true.

The results of the rest of the examples can be determined in a similar manner, then tested on the computer.

PROGRAMMING EXAMPLES

The examples in this chapter are intended for the first time user. They explain in step by step detail the procedures for creating, editing, and saving a BASIC program. If a more detailed description of a BASIC instruction is required, the reader is referred to the chapters of this manual which cover the instructions on an individual and in depth basis.

It is recommended that the novice BASIC user go through this section while seated in front of a terminal, with BASIC up and running. This way, each instruction and program can be tried out as it is presented.

When BASIC is loaded into a minimum disk system configuration with 32K bytes of memory, the space available to the user should be 1.5K bytes. The other 30.5K bytes are occupied by BASIC, the console processor, CDOS I/O routines, etc. For ways to increase the user space, see the notes at the end of the manual, specifically: Changing the Number of I/O Channels.

Note: It is important to remove disks from the disk drive before turning the power on or off. This will eliminate the possibility of stray bits being written on the disk as the system is powered up or down.

7.1   Getting Started

After the computer has been turned on, the floppy diskette containing BASIC and the Cromemco Disk Operating System (CDOS) must be inserted into disk drive A. This is the leftmost disk drive. Start with the diskette supplied by Cromemco. The diskette should be inserted so that the label faces left and the edge with the elongated hole which exposes the surface of the diskette is toward the

rear of the machine (disk drive). If there is a
door on the disk drive it must be closed after the
diskette is inserted.

Next the RETURN key on the console must be
depressed several times so that the computer can
determine the speed at which the console is sending
characters (baud rate). The computer will respond
to this by displaying a character on the console
(called a prompt) which indicates that it is
waiting for the user to type something. If the
prompt is a semicolon (;), the user must type B
(for Boot) followed by a carriage RETURN to load
CDOS into memory. If the prompt is the letter A
followed by a period (A.) then the Cromemco Disk
Operating System (CDOS) has already been loaded
into memory.

The terminal must be set up so that it will only
send BASIC upper case characters. If you are using
one of the Cromemco 3100 series terminals, depress
the ALPHA LOCK key so that the red light on the key
is lit. You will now be in upper case mode.

In response to the CDOS prompt, type BASIC followed
by a carriage RETURN. After a few seconds the
BASIC sign on message will be displayed followed by
the BASIC prompt (>>). BASIC has now been loaded
into the computer's memory and is waiting for
instructions.

One Very Important Point follows:

ALL LINES ENTERED IN BASIC MUST BE TERMINATED BY
DEPRESSING THE CARRIAGE RETURN KEY.

BASIC will never respond to an instruction, accept
any input, or complete a command unless it is
terminated by a carriage RETURN. If BASIC does not
seem to be responding as you think it should, make
sure that you have properly terminated the current
line (with a carriage RETURN).

7.1.1   The Command or Immediate Mode

Whenever the BASIC prompt (>>) is displayed, BASIC is in the Immediate Mode. This means that an instruction (command) can be entered and BASIC will respond immediately after the carriage RETURN is depressed.

The first instruction we will discuss is PRINT. PRINT causes BASIC to display the information following the word PRINT. Using only the PRINT instruction, and the standard Arithmetic Operators (+ for addition, - for subtraction, / for division, and * for multiplication), BASIC may be used as a calculator.

Instruction	Explanation
PRINT 5	This will PRINT the number 5. Since there is nothing following the 5 BASIC will add a carriage RETURN and LINE FEED so that the next item which is PRINTed will start in column 0 of the following line.
PRINT 7 + 4	This will PRINT the result of adding 7 and 4, or 11.
PRINT "FRED"	This will PRINT the word FRED. Notice that if a group of letters (called a string) are to be PRINTed, they must be enclosed in quotation marks. When a string is enclosed in quotation marks, it is called a string literal. Numbers (called constants) which are to be used in a computation may not be enclosed in quotation marks.
PRINT	This will PRINT a blank line.
PRINT 1,2,3,4	This will PRINT the numbers 1 through 4 in four columns across the screen. When commas (,) are used to separate items in a PRINT list, the items are aligned in four columns across the console screen. If

                           semicolons (;) are used, the
                           items are displayed with no
                           intervening spaces.

PRINT "ANS. = ";75   This will display ANS. = 75 on
                      the console. Notice that a
                      blank was included after the
                      equal sign in the string
                      literal. This blank is
                      considered part of the string
                      literal, just as any other
                      character.

Next we will discuss variables and the Assignment
Operator (Instruction). A variable is the name of
a location in the computer's memory. While the
name of a variable stays the same, the contents of
the variable (location) may vary.

A variable name may be thought of as a label which
has been affixed to a box. If we are speaking of
an arithmetic variable, then the box would contain
a number. The number inside the box is the VALUE
of the variable. The value of the variable can be
changed while the name of the variable stays the
same. Similarly, a string variable would contain a
string of characters which could include letters,
spaces, numbers, and any other printable
characters. This combination of characters would
be the value of the string variable.

Arithmetic variable names are composed of either a
single letter (A-Z) or a single letter followed by
a single number (Ø-9). Some examples of arithmetic
variables are A1, B8, ZØ, Q4, V, A, etc.

String variables may use any name which is a legal
arithmetic variable name, however, a dollar sign
must immediately follow the name. Some examples of
string variables are G8$, Y$, P$, T1$, M9$, etc.

The assignment operator assigns a value to a
variable. The equal sign (=) is used for this
purpose. In BASIC, the equal sign can be read as,
"is assigned the value of". Notice that it does
not necessarily denote equality.

Instruction	Explanation
G$ = "FRED"	The value of the string literal "FRED" is assigned to the string variable named G$. Notice that, although the string literal "FRED" must be enclosed in quotation marks, the quotation marks are not part of the string variable. The quotation marks indicate to BASIC that the enclosed characters are to be considered a string literal.
PRINT G$	This instruction will display the value of G$ on the console. If this instruction follows the preceding one, FRED will be PRINTed on the console terminal.
P4 = 775	The variable named P4 is assigned the value of 775.
P4 = P4 + 1	P4 is assigned the value of P4 + 1. If this instruction follows the previous one, P4 will have the value of 776.
Q5$ = " IS NO. "	The string variable Q5$ is assigned the value of the string literal " IS NO. ".
Z7$ = Q5$	The string variable Z7$ is assigned the value of the string variable Q5$.
PRINT G$; Z7$; P4	will print FRED IS NO. 776 (assuming it follows the above assignment instructions).

An important distinction: The user must be careful to distinguish between a string literal and the name of an arithmetic variable. The following table should help to clarify this difference.

Instruction	Explanation
A = 54	The variable named A is assigned the value of 54.
PRINT A	This will PRINT the number 54 which is the value of the arithmetic variable A.
PRINT "A"	This will PRINT the letter A which is the value of the string literal.

The next instruction we are going to discuss is the INPUT instruction. When executed, this instruction displays a question mark on the console and waits for the user to supply a number or a string.

Instruction	Explanation
INPUT A	This command will display a question mark and wait for the user to enter a number. The number which is entered will be assigned to the variable named A.
INPUT Q$	This command will display a question mark and wait for the user to enter a string (any group of characters, in this case up to a maximum of eleven). The value of the string which is INPUT will be assigned to the string variable named Q$.

7.1.2   The RUN or Program Execution Mode

Up to this point, all of our examples have been executed as commands or in what is referred to as the immediate mode. Each command was executed as soon as the carriage RETURN was depressed. Once executed, the entire instruction had to be typed again in order to be re-executed.

Now we shall use these same instructions to write a program. A program consists of statements. A

statement is nothing more than an instruction with
a line number preceding it. Line numbers are also
called statement numbers; the two terms are
interchangeable. The line number indicates to
BASIC that the instruction is to be stored in
memory for later execution. A line number is also
a handy way to refer to a BASIC statement if it
needs to be changed.

Let us enter a program:

        >>100 INPUT A$
        >>200 PRINT A$; " is smart!"

This program is now current in the User Area. The
User Area is the BASIC workspace in which a program
can be written, edited, and run. The LIST command
displays the contents of the User Area.

The RUN command causes BASIC to execute the program
in the User Area. Program execution begins with
the statement with the lowest line number and
continues sequentially.

Now, we can execute the program:

        >>RUN
        ? Tom   BASIC displays the ? (prompt)
                           and the user enters a string.
                           Don't forget the carriage
                           RETURN at the end of the
                           string!

        Tom is smart!   BASIC   PRINTs   the   string
                           variable (Tom) and the string
                           literal (is smart!).

        END   BASIC tells you that it's done
                           with the program,

        >>   and that it is ready for
                           additional instructions.

To get a LISTing of your program, type the command
LIST in response to the BASIC prompt.

        >>LIST
          100 INPUT A$
          200 PRINT A$;" is smart!"

        >>

65

7.1.3   Activating the System Printer

The printer (if one is attached to your system) is
activated by typing control-P. Control-P is typed
by holding down the CTRL key on your terminal (as
you would hold the shift key on a typewriter) and
simultaneously depressing the P key.

Anytime CTRL-P is typed, the system printer
automatically echoes any input, output, or LISTing
which is sent to the console.

To LIST the program which is current in the User
Area, enter:

    >>LIST (CTRL-P) (CR)

on the console. The LISTing will be sent to the
console as well as the printer. When the LISTing
has terminated, type CTRL-P before anything else is
entered. This will ensure that nothing else is
output to the printer following the LISTing.

Anytime the printer is on and is echoing the
console, typing CTRL-P will turn off the printer.

If CTRL-P is inadvertently typed, and there is no
system printer, or the printer is off, type a
second CTRL-P to resume normal operation. If this
is not done, the system will wait for a printer
before displaying any additional output on the
console.

7.1.4   Program Editing

Continuing with the same program, we wish to alter
a statement.

To change a line, type it over again. When BASIC
recognizes that it already has a line with the same
line number, it will replace the old line with the
new one.   To delete a line, type the line number
followed by a carriage RETURN. Suppose we wanted
to change line 200:

>>200 PRINT A$; " is very smart!!!"

>>LIST   After making the change, we can
                list the program to make sure
                that we changed it properly.

  100 INPUT A$
  200 PRINT A$;" is very smart!!!"

>>   Again, BASIC waits for further
                instructions.

Suppose that, instead of executing these statements
only once, we wanted them executed several times.
We could type RUN each time the program was to be
executed, or we could add another statement which
would direct BASIC back to the beginning of the
program each time it finished PRINTing:

>>300 GOTO 100 This GOTO tells BASIC to GOTO
                statement number 100 and
                continue executing from there.
>>LIST
  100 INPUT A$
  200 PRINT A$;" is very smart!!!"
  300 GOTO 100

>>RUN
? Alice   Don't forget the carriage
                RETURN!

Alice is very smart!!!

? Eileen   After it was done with
                statement 200, statement 300
                told BASIC to go back to
                statement 100, which gave us
                another question mark.

67

Eileen is very smart!!!

?   This could (and would) go on
                     forever. This can be avoided
                     by depressing the ESCape key
                     (appropriately named). On a
                     Cromemco terminal this key is
                     located on the upper left of
                     the keyboard and is marked
                     ESC.

100 ESCAPE
                     BASIC tells you what happened,
>>   and waits for further
                     instructions.

7.2   Example Program One

Let's write a program to compute a person's age in
the year 1985. For this program, we will introduce
two new instructions.   The first of these is SCR
which is short for SCRatch.   This instruction
clears memory of a program or any statements which
may be in the User Area:

>>SCR

>>LIST   Nothing will be listed, because
                  everything has been SCRatched!

>>

The other is AUTOL which is short for AUTOmatic
Line numbering.   This is a convenient feature of
Cromemco 16K BASIC which allows the programmer to
concentrate on the program and forget about
entering line numbers.   AUTOL is followed by two
numbers (called arguments).   The first of these
indicates the first line number, while the second
indicates the increment between line numbers.   In
the following example, the line numbers are
automatically typed by BASIC while the program is
entered by the user:

>>AUTOL 1000,10

>>1000 PRINT"Enter the year of your birth: ";
>>1010 INPUT Y
>>1020 PRINT"In 1985 you will be ";1985-Y;" years old"
>>1030 PRINT
>>1040 GOTO 1000
>>1050
>>

There are several things about this program which
bear discussion.   The first is that the AUTOmatic
Line numbering mode may be terminated by entering a
carriage RETURN (our old friend) in response to a
line number. Line 1020 PRINTs two strings with a
number in the middle. The number is the result of
a computation: the numeric variable Y is
subtracted from the numeric constant 1985.   Line
1030 is included so that there will be a blank line
PRINTed between examples.

7.2.1   LISTing to a Disk File

The program we have written will stay in memory (in
the User Area) as long as we don't execute the
SCRatch command or turn the power off.

We may write the program to a disk file so that it
may be retrieved at a later time. Once it has been
LISTed on the disk, we may SCRatch the user area or
turn the power off and our program will still be
safe on the disk.

The only way a disk file can be destroyed is by
ERASing it or by writing another file with the same
name to the disk. In the latter case, the second
file would be written over the first so that the
first file would be destroyed.

The LIST command, as we have been using it, LISTs
the contents of the User Area to the console. If
the LIST command is followed by a string literal,
BASIC will interpret the string literal as a file
name, and LIST the contents of the user area to
that file. Assuming that our program is still in
the User Area, the following command will write the
program to a file called FIRST.

>>LIST "FIRST"

>>

The disk should come to life, click once or twice,
and another prompt should appear on the console.
If BASIC has done your bidding, the DIRectory
command will reveal a file on the disk called
FIRST. To get a list of all the files on the disk
enter the command:

>> DIR

BASIC will respond with a list of file names,
including FIRST. Now you can shut off the computer
without losing your program.

Remember to remove the disk before turning the
power off. It is also a good idea to insert the
disk only after the power has been turned on. Do
not turn the power on or off while the disk is
inserted in the machine.

7.2.2   ENTERing from a Disk File

A program which has been LISTed to a disk file may
be re-entered by using the ENTER command.

Before entering any program, whether from a disk
file, or from the console, it is a good idea to
SCRatch the User Area. Assuming that we are once
again (or still) in BASIC, we can give the
following two commands to clear the User Area and
enter our program which was saved in the disk file
FIRST.

        >> SCR

        >> ENTER "FIRST"

        >>

Our program may now be LISTed, RUN, or modified as
we desire.

7.3   Example Program Two

Let's try another example. First we will type SCR
to clear the user area, and then AUTOL to start the
AUTOmatic Line numbering:

>> SCR

>> AUTOL 10,10

Next, type in your first instruction:

>>10 LET N = 5

When the program is RUN, this instruction will
assign a value of 5 to the variable N. BASIC will
remember this value and it will remain the same
until redefined by the user or by an algorithm
within the program. BASIC will also understand the
statement above without the LET, as:

>>10 N = 5

This is called an implied LET and is a
characteristic of advanced versions of BASIC.
BASIC will also disregard spacing of characters in
statement lines. Thus, the statement:

10 LET N = 5

is equivalent to the statement:

10 LET N=5

At this point, we have the following line on the
terminal:

>>AUTOL 10, 10

>>10 N = 5

BASIC is waiting for our next instruction. Type
in:

>>20 INPUT A: INPUT B: INPUT C: INPUT D: INPUT E

With this long statement line, we have created a
set of instructions that will ask for five numbers
(inputs A,B,C,D, and E) when the program is run.
In 16K BASIC, more that one instruction can be
associated with a line number. These instructions

72

must be separated by colons. Multi-instruction
statement lines save memory space.

Next, we enter the statement:

>>30 X = (A+B+C+D+E)/N

This assigns the value of the sum of the five INPUT
numbers divided by the value of the variable N to
variable X. The variable X is therefore the
average of these five values. We defined N as 5
previously. If we had not defined N, then it would
be assumed that N = Ø.

Naturally, we'll want to look at X to see what it
is. We type in:

>>40 PRINT X.

which instructs BASIC to print that value of X when
the program is run. What do we have so far? Our
program, as listed below, looks ready to run.

```
>>AUTOL 10, 10

>>10 N = 5
>>20 INPUT A: INPUT B: INPUT C: INPUT D: INPUT E
>>30 X = (A+B+C+D+E)/N
>>40 PRINT X
```

16K BASIC does not require an END or STOP, but it
is good programming practice to include one. So we
can add the following statement to our program:

>>50 END

If we wanted to run through the program a number of
times, finding means for many groups of five
numbers, we might have entered:

50 GOTO 10

This would instruct BASIC to return to line 10 and
re-execute the program, finding the mean of five
new numbers each time. To escape from such a loop,
or from a program during its execution, press the
ESCape key on your terminal or teletype keyboard.

Now let's run the program. After statement line 50
is typed in we depress the carriage RETURN in
response to the next line number to get out of the

AUTOmatic Line numbering mode. Then we type in:

>>RUN

This command instructs the computer to begin
execution of the program. When the first INPUT
command in line 20 is encountered during program
execution, a question mark (called a prompt) will
appear on your terminal, as indicated below:

>>RUN
?

This prompt indicates that the computer is waiting
for you to input a number for the first variable in
the INPUT statement. We therefore type in:

? 17

and BASIC responds with:

? 17
?

The second prompt indicates that the computer is
waiting for a value for INPUT B. In fact, we need
to enter values for the next four INPUT variables.
Once we input these values and depress the carriage
RETURN, the computer will complete program
execution and PRINT out the value of X, as shown
below:

>>RUN
? 17
? 25
? 12
? 18
? 13
17
50 END

Almost instantly, the BASIC program has calculated
and printed the average of the five numbers we
input. This average is 17. At this point, we
might want to do a little formatting to change the
way in which the results are printed out. We LIST
the program again to see what we have to work with.

```
>>LIST

10 N = 5
20 INPUT A: INPUT B: INPUT C: INPUT D: INPUT E
30 X = (A+B+C+D+E)/N
40 PRINT X
50 END
```

We can start by inserting some lines within the
present text of the program. At any time, lines
can be inserted between present statements by using
a number which falls between two line numbers. For
example, a line numbered 15 will be inserted
between statement number 10 and statement 20. When
a program is RUN or LISTed, the new statement lines
are automatically inserted into the proper space in
the hierarchy.

We decide to add spaces between lines to make
reading easier. We add:

```
>>15 PRINT
>>16 PRINT
>>35 PRINT
>>36 PRINT
```

Inserting a PRINT instruction in a statement line
without text to be printed generates a linefeed.
At this point, you'll want another look at the
program. BASIC will have sandwiched the new
statement lines into the old text as shown below:

```
>>LIST

10 N = 5
15 PRINT
16 PRINT
20 INPUT A: INPUT B: INPUT C: INPUT D: INPUT E
30 X = (A+B+C+D+E)/N
35 PRINT
36 PRINT
40 PRINT X
50 END
```

Now type RUN. According to our program, the
computer will skip two lines after you type in RUN
and then wait for you to type in 5 numbers. Once

these numbers are input, the program instructs the
computer to skip two more lines and then PRINT the
result. This output appears on the terminal as
follows:

>>RUN

? 22
? 14
? 16
? 20
? 28

20
50 END

Many more variations may be added. For example,
the statement:

5 PRINT "A PROGRAM TO CALCULATE THE"
7 PRINT "AVERAGE OF 5 NUMBERS"

will give you a title for this program. You might
also replace line 40 with:

40 PRINT "THE MEAN OF THESE NUMBERS IS ";X

If it becomes necessary to enter a line between two
others whose line numbers are consecutive (say you
need a line between lines 35 and 36), the RENUMBER
command can be used to spread the line numbers
apart. While a program is in the User Area, type
the command:

>> RENUMBER

>>

and BASIC will assign new line numbers starting
with 10 and incrementing by 10.

7.3.1   Using the SAVE and LOAD commands

The SAVE command will write the program which is
current in the User Area to a disk file.   The
program will be in internal machine format unlike a
program which is written to disk using the LIST
command.   The LOAD command is to the SAVE command
what the ENTER command is to the LIST command.
LOAD allows the user to read a SAVEd file into the
User Area.   Notice that LOAD cannot be used with a
LISTed file and that ENTER cannot be used with a
SAVED file.

The formats of these commands are:

>>SAVE "SECOND"
>>LOAD "SECOND"

When used with larger programs, SAVE and LOAD can
be significantly faster than LIST and ENTER.   Also,
when the RUN instruction is given with the name of
a SAVEd file, the specified file will be LOADed and
RUN with only that one instruction.   Programs may
be chained in this fashion, one calling the next,
calling the next, etc.

>>RUN "SECOND"

or

9999 RUN "SECOND"

The execution of either of these instructions will
cause the SAVEd program SECOND to be LOADed into
the User Area and execution to begin.

7.4   Example Program Three

Active Bandpass Filter Calculation

One of the most elementary applications for a
computer is to use the computer to do calculations
that can be done on sophisticated hand calculators.
A program written to perform such calculations can
be saved and reused as often as required. An
example of a simple calculation program follows.
This program allows the user to input data on
certain parameters associated with the design of an
active bandpass filter based on a common
operational amplifier. The program then calculates
the necessary component values based on the data.

A BANDPASS 741-TYPE ACTIVE FILTER
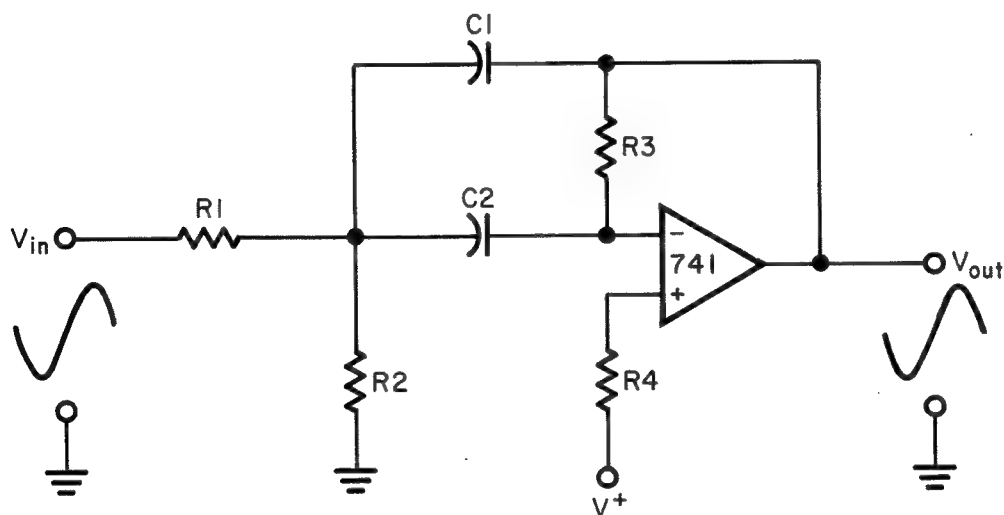
NOTE: NO PHASE RELATIONSHIP IMPLIED

78

The first three statement lines of the program are
as follows. Notice that the at sign (@) is used
instead of the word PRINT. When a lot of PRINT
instructions are used, this can facilitate program
entry.

```
 10 @
 20 @"THIS PROGRAM CALCULATES DATA NEEDED TO CONSTRUCT"
 30 @"ACTIVE BANDPASS FILTERS USING 741-TYPE OP-AMPS."
```

The principal function of PRINT statements 20 and
30 is to document the program so that it can be run
in the future by users who are unfamiliar with it.
All programmers should be encouraged to document
programs thoroughly.

The next step in the program is to request the user
to input a number of values for various parameters.
Note that each variable is defined in preceding
PRINT statements.

```
 40 @
 50 @"WHAT IS THE CENTER FREQUENCY OF THE PASS BAND,"
 60 @"IN HERTZ (E.G., 4000, 250, 60)";
 80 INPUT F
 90 @
100 @"WHAT IS THE DESIRED GAIN, IN DECIBELS?"
110 @"(E.G., 0, 5, 25)";
130 INPUT H
140 @
150 @"WHAT IS THE DESIRED Q OF THE FILTER?";
170 INPUT Q
180 @
190 @"SELECT A CONVENIENT STARTING VALUE C FOR"
195 @"CAPACITORS C1 AND C2."
200 @"IF THE VALUE IS IN PICOFARADS, ENTER THE DATA"
210 @"IN THE FORMAT: X...E-12. IF THE VALUE IS"
220 @"IN MICROFARADS, USE THE FORMAT:  X...E-6."
240 INPUT C
```

In this schematic formula, C1 and C2 are equal.
Once a value for C is specified, the resistor
values can easily be found. The next step then is
to specify the formulas for each resistor value, as
indicated below:

```
250 @ : @ : @
280 @"RESISTANCE IN OHMS"
290 @ : @
310 LET W=F*2*3.14159
320 @ USING"R1 = #########.###",Q/(H*W*C)
```

```
330 @ USING"R2 = #########.###",Q/(((2*Q*Q)-H)*(W*C))
340 @ USING"R3 = #########.###",(2*Q)/(W*C)
350 @ USING"R4 = #########.###",2*((2*Q)/(W*C))
```

These formulas for R1, R2, R3, and R4, specify the
calculations which must be performed to determine
the various resistances. The final section of the
program provides a simple routine which allows the
user to easily re-enter the program and input a new
set of values.

```
360 @ : @
380 @"WOULD YOU LIKE TO CALCULATE ANOTHER FILTER?"
390 @"TYPE Y FOR YES";
410 INPUT A$
420 @ : @ : @
450 IF A$(Ø,Ø)="Y"THEN GOTO 40
460 END
```

If the user does not wish to do another filter
calculation and consequently types in a string
starting with a character other than Y, then the
program ends and BASIC returns to the immediate or
command mode.

Note that the variables in this program are given
letter symbols that correspond to their function.
For example, it is surely easier to remember that
the Q of the filter is represented by variable Q
than by variable L. Similarly the variable for
capacitors C1 and C2 is C, rather than X or Y.
These may seem to be intuitively obvious at this
level of complexity, but as the program complexity
increases the assignment of appropriate variable
names becomes much more difficult.

Once the program has been input, we can instruct
the computer to execute the program by typing in
RUN. The following output will be generated:

>>RUN

THIS PROGRAM CALCULATES DATA NEEDED TO CONSTRUCT
ACTIVE BANDPASS FILTERS USING 741-TYPE OP-AMPS.

WHAT IS THE CENTER FREQUENCY OF THE PASS BAND,
IN HERTZ (E.G., 4000, 250, 60)? 1000

WHAT IS THE DESIRED GAIN, IN DECIBELS
(E.G., 0, 5, 25)? 20

WHAT IS THE DESIRED Q OF THE FILTER?? 100

SELECT A CONVENIENT STARTING VALUE C FOR
CAPACITORS C1 AND C2.
IF THE VALUE IS IN PICOFARADS, ENTER THE DATA
IN THE FORMAT: X...E-12. IF THE VALUE IS
IN MICROFARADS USE THE FORMAT: X...E-6.
? 1.0E-6

RESISTANCE IN OHMS

R1 =   795.775
R2 =   .005
R3 = 31831.016
R4 = 63662.031

WOULD YOU LIKE TO CALCULATE ANTOHER FILTER?
TYPE Y FOR YES? NO

460 END

7.5   Example Program Four

Statistical Analysis Program

Another common use for a computer is the
calculation of statistics for a given set of data,
particularly very large data sets. The program
listed below calculates six common statistics for a
given data set.

One of the first steps that must be taken in any
program is the assignment of variable names to any
variables which will appear in the program. The
following program involves six functions which will
require variable names. These functions and the
corresponding variable name used in the program to
represent each function are listed below:

Variable Name	Function
N	Number of elements in a data set
S	Sum of the numbers in a data set
T	Sum squares
M	Mean
V	Variance
D	Standard Deviation

We instruct the computer to keep track of the
number of elements in a data set by including a
counter variable in the program. In this case, the
counter variable is N. To count the number of
elements, we set N equal to Ø initially and then
increment N by 1 each time a new element in the
data set is READ. Consequently, the first
statement lines are used to initialize (set equal
to Ø) certain variables.

```
10 LET N=Ø
20 LET S=Ø
30 LET T=Ø
```

Note that S and T are also set equal to Ø. This
will allow us to sum data elements and to sum the
squares of data elements as the elements are READ.
We can now begin reading in data elements.

82

```
       40 READ X
       50 IF X=999 THEN GOTO 100
       60 LET N=N+1
       70 LET S=S+X
       80 LET T=T+(X*X)
       90 GOTO 40
      100 IF N=0 THEN GOTO 300
```

The value 999 is used as a dummy value to indicate
the end of a data set. The program continues to
read in data and increment the values of the
variables N, S, and T until a 999 is read from a
DATA statement. Once a 999 value is read in,
program control is transferred to statement line
100.

Statement line 110 through 130 compute the
remaining three statistics.

```
      110 LET M=S/N
      120 LET V=(N*T-S*S)/N/(N-1)
      130 LET D=SQR (V)
```

Next we include a set of titles to be printed out
with each statistic.

```
      140 PRINT
      150 PRINT
      160 PRINT "NUMBER","SUM","SUM SQUARES"
      170 PRINT N,S,T
      180 PRINT
      190 PRINT "MEAN","VARIANCE","STANDARD DEVIATION"
      200 PRINT M,V,D
      210 PRINT
      220 PRINT
```

At this point, all the statistics have been
calculated for a set of data. Program control is
therefore returned to the first statement in the
program so that the next set of data can be read.

```
      250 GOTO 10
      260 DATA 1,2,3,4,5,6,7,8,9,10,999
      270 DATA 2.3,2.4,2.5,2.6,2.7,2.8,999
      280 DATA 10,15,29,28,12,44,5.5,2.2,999
      290 DATA 999
      300 END
```

Once this program has been entered, edited, and
examined for errors, it can be RUN. The results
are as follows:

```
>>RUN
```

NUMBER	SUM	SUM SQUARES
10	55	385

MEAN	VARIANCE	STANDARD DEVIATION
5.5	9.16666666666	3.0276503540975

NUMBER	SUM	SUM SQUARES
6	15.3	39.19

MEAN	VARIANCE	STANDARD DEVIATION
2.55	0.035	0.18708286933869

NUMBER	SUM	SUM SQUARES
8	145.7	4065.09

MEAN	VARIANCE	STANDARD DEVIATION
18.2125	201.64696428571	14.200245219211

```
***300 END***
```

7.6 Example Program Five

```
10 REM This is a program which demonstrates
20 REM the implementation of a Random
30 REM Access File.
40 REM
50 REM The program creates and opens a file, writes
60 REM 51 records sequentially, allows
70 REM the user to repeatedly select any
80 REM of the 51 records at random, and
90 REM then closes and erases the file.
100 REM
110 DIM A$(21)
120 INTEGER I,J
130 CREATE"RANDTEST"
140 OPEN\1,24\"RANDTEST"
150 ON ERROR GOTO 220
160 ON ESC GOTO 320
170 PRINT"Pause, writing file RANDTEST"
180   FOR I=0 TO 50
190   PUT\1\"This is record number ",I
200   NEXT I
210 PRINT
220 PRINT"Which record would you like to see?"
230 PRINT"Enter record number (0-50) or -1 to stop: ";
240 INPUT I
250 IF I<0 THEN GOTO 320
260 IF I>50 THEN GOTO 210
270 GET\1,I\A$(-1),J
280 PRINT"Contents of record number ";I;" is:"
290 PRINT A$;J
300 PRINT
310 GOTO 230
320 CLOSE
330 ERASE"RANDTEST"
340 PRINT : PRINT : PRINT"File RANDTEST erased"
350 END
```

On line 130, the file RANDTEST is CREATEd. If a
file does not exist in the directory, it must be
CREATEd before it can be OPENed for reading or
writing. The file is OPENed on line 140 with a
record length of 24 bytes which are allocated as
follows:

     22 bytes for the string, and
     2 bytes for an integer

Line 150 ensures that any run time error (such as
an invalid user response) will return control to

85

the user <u>within</u> the program and will not cause the
program to terminate abnormally. Because many
programs can be terminated by the use of the ESCape
key, statement 160 is included in this program.
The program will still be terminated by depressing
the ESCape key, but the active file will be closed
and deleted before control is returned to the user.

Statements 180 through 200 write out 51 records,
each containing (for identification) the record
number in addition to a string. Because no record
numbers are specified, the records are written
sequentially starting with record zero.

The user is then asked for the number of the record
to be displayed and on line 270 the record whose
number is specified by the variable I is retrieved.
Because the records which were written to this file
each contained a 22 character string followed by an
integer number, they must be read back into
variables of the same type and length.

The user is allowed to view as many records as are
desired. Entering a negative one (-1) when asked
for a record number will cause the file to be
CLOSEd and ERASEd and program execution to be
terminated.

INSTRUCTIONS AND FUNCTIONS

The following chapters describe the instruction set
of Cromemco's 16K Extended BASIC. Although several
of the instructions perform a variety of tasks, an
attempt has been made to group the instructions and
functions into logical divisions.

Program Development Instructions encompass those
instructions which are most frequently used by a
programmer while developing or modifying programs.
Documentation covers the REMark instruction and
Assignment covers the use of the Assignment
Operator in the LET and MAT instructions.

The Initialization chapter describes those
instructions which set and change the type of
numeric representation which is used in a program
as well as the DIMension instruction.

Control Structures are the instructions which
control the logical flow of a program. This
includes conditional and unconditional transfer of
control, loops, and program termination.

The Input/Output chapter begins with instructions
which may be used to transfer information to and
from the console terminal as well as those
instructions which are used to read DATA statements
from the program itself. A discussion of the
theory and use of Data Files precedes the
definition of the various 16K BASIC instructions
which allow the user to control and access files.

The next chapter covers Arithmetic, Trigonometric,
Programmer Defined, and String Functions.

The System and File Status chapter shows the user
how to determine and change various system statuses
and parameters.

Then Machine Level Instructions which allow the
user to interact with Assembly Language subroutines
and Fatal and Non-fatal Error Messages are

discussed.

The final chapters include a Glossary of words
which may be unfamiliar to the novice, and an
Appendix which describes various aspects of the
Language which may be of interest to the advanced
user.

instruction: AUTOmatic Line numbering

format: AUTOL n,m

where:

n   is the starting line number.

m   is the line number increment.

The AUTOL command provides automatic statement line
numbering so that the user does not have to enter a
line number for each line when entering a program.

Note:

1.   The automatic generation of line numbers may
     be terminated by pressing the ESCape or
     carriage RETURN key when the user is prompted
     for the next line.

Example:

```
>>AUTOL 100,10

>>100 REM ALL LINE NUMBERS IN THIS
>>110 REM EXAMPLE ARE GENERATED BY BASIC.
>>120 I,=5
        $
ERROR 1 -- SYNTAX
>>120 I=5
>>130 REM NOTICE THAT AFTER A SYNTAX ERROR
>>140 REM AUTOL WILL RE-PROMPT SO THAT
>>150 REM THE ERROR CAN BE CORRECTED.
>>160 PRINT I
>>170 END
>>180 (carriage RETURN)
>>
```

instruction: BYE

format: [Ln] BYE

where:

Ln   is an optional number.

The BYE instruction is used to exit from BASIC and
return to CDOS on disk systems or MONITOR on non-
disk systems.

Notes:

1.   For disk systems, after the BYE command is
     typed in, the computer will respond with the
     current disk drive.

2.   For non-disk systems, BYE goes to location
     E008H in the MONITOR.

3.   See the Appendix, Areas of User Interest, for
     information on how to cause BYE to jump to
     another address.

4.   BYE will close all files which are OPEN at the
     time the instruction is executed.

Example:

     >>BYE

     B.

In this example, the user has typed the BYE
command. The next prompt displayed (B.) indicates
that the user is in CDOS and that B is the current
drive.

instruction: DELETE

format: [Ln] DELETE n

[Ln] DELETE n, m

where:

Ln   is an optional line number.

n   is the statement number of the
             first or only line to be
             DELETEd.

m   is a statement number. If used,
             it indicates that lines numbered
             n through m are to be DELETEd.

The DELETE instruction is used to remove statement
lines from the program currently in the User Area.

Notes:

1.   The DELETE instruction must be the last (or
     only) instruction on a line.

2.   The DELETE instruction must have at least one
     argument.

Example:

    >>LIST

        10 INPUT A,B,C
        20 D = A+B+C
        30 PRINT A
        40 PRINT B
        50 PRINT C
        60 PRINT D
        70 END

    >>DELETE 30,50
    >>LIST

        10 INPUT A,B,C
        20 D = A+B+C
        60 PRINT D
        70 END

Here the DELETE command removes lines 30 through 50
from the program.

command: DIRectory

format: DIR

DIR svar

where:

svar   is a string variable or a string
            literal file reference.

The DIR command corresponds to the CDOS DIR command
(see the CDOS User's Manual for a full
description). The DIR command lists disk files
giving size (in K-bytes) and number of extents. If
the optional file reference is used, it must be
enclosed in quotation marks (string literal) or
else must be a valid string variable.

Examples:

DIR   will list all files on the
                 current disk.

DIR "A:*.*"   will list all files on drive A.

DIR "*.SAV"   will list all files on the
                 current disk with the extension
                 SAV.

instruction: ENTER

format: [Ln] ENTER svar

where:

Ln   is an optional line number.

svar   is a string variable or string
          literal file reference.

The ENTER instruction is used to ENTER a BASIC
program (in ASCII format) from a disk file or other
external device into the User Area.

Notes:

1.   ENTER will read a LISTed program into the User
      Area. ENTER will not read a SAVEd file.

2.   ENTER does not delete statement lines from the
      program which is currently in the User Area.
      ENTER does replace lines in the current
      program with lines from the file being ENTERed
      if the line numbers are the same.

3.   Efficient use of memory following an overlay
      results if the ENTERed program, to as great an
      extent as is possible, replaces lines in the
      current program and does not add new line
      numbers.

4.   The ASCII ESCape character (1BH) is used as
      the end of file mark by BASIC. ENTER looks
      for this character to determine the end of the
      program.

instruction: LIST

    format: [Ln] LIST

              [Ln] LIST n

              [Ln] LIST n,m

              [Ln] LIST svar

              [Ln] LIST svar,n

              [Ln] LIST svar,n,m

where:

    Ln   is an optional line number.

    svar   is a optional string variable or
              a string literal file reference
              denoting the destination of the
              LISTing. If omitted, the
              LISTing will go to the console.

    n   is the line number of the first
              line to be LISTed. If omitted,
              the entire program will be
              LISTed.

    m   is the line number of the last
              line to be LISTed. If omitted,
              the program will be listed
              through the last line.

The LIST instruction is used to LIST one or more
statement lines from the User Area to the console,
a disk file, or another file device in ASCII
format. The instruction may be used to output an
entire program, a block of statement lines within a
program, or a single statement line. The formats
of LIST which do not use svar will direct the
output to the console.

Notes:

1.   A program which has been LISTed to a disk can
     be read back into the User Area using the
     ENTER instruction. It can <u>not</u> be read back
     using LOAD or RUN.

2.   LISTed files are compatible between different

94

versions of Cromemco 16K Extended BASIC as
well as different versions and sizes of the
Cromemco Disk Operating System (CDOS). SAVEd
files are not necessarily compatible in this
manner.

3.   The ASCII ESCape character (1BH) is used as
the end of file mark by BASIC. LIST outputs
this character at the end of a program which
is sent to the disk.

instruction: LOAD

format: [Ln] LOAD svar

where:

Ln   is an optional line number.

svar   is a string variable or string
          literal file reference.

The LOAD instruction is used to LOAD a BASIC
program (in internal machine format) from a disk
file into the User Area.

Notes:

1.   LOAD will read a SAVEd program into the User
     Area. The program must have been SAVEd under
     the version of BASIC and the version and size
     of the Cromemco Disk Operating System under
     which BASIC is currently being run.

     LISTed files are compatible between different
     versions of Cromemco 16K Extended BASIC as
     well as different versions and sizes of CDOS;
     SAVEd files are not.

2.   The LOAD instruction resets (clears) all
     variables, string variables, and matrices.

3.   LOAD resets the trigonometric mode to RADians.

instruction: RENUMBER

   format: RENUMBER

       RENUMBER m

       RENUMBER m,n

       RENUMBER m,n,b

       RENUMBER m,n,b,e

where:

  m   is the starting line number in the RENUMBERed program.

  n   is the line number increment in the RENUMBERed program.

  b   is the first existing line number to be RENUMBERed.

  e   is the last existing line number to be RENUMBERed.

The RENUMBER instruction alters the statement numbers in the current program.

Notes:

1. The default value for the RENUMBER instruction is a starting line number (m) of 10 and an increment value (n) of 10.

2. If only the first parameter (m) is specified, the second parameter (n) assumes the same value. In the example below, the command RENUMBER 100 is equivalent to the command RENUMBER 100,100.

3. The RENUMBER instruction alters line numbers imbedded in the entire program in GOTO, GOSUB, and IF-THEN statements to conform to the RENUMBERed statements. This will affect a line which is not RENUMBERed if the line contains a reference to a RENUMBERed line.

4. RENUMBER can not normally be used to re-order or rearrange sections of a program relative to other sections. If line numbers are LISTed

out of order after the RENUMBER instruction is given, follow the procedure in the following note (5a-d) to rearrange the lines into numeric order.

5.   The RENUMBER command will include DELETED statement numbers in the sequence of renumbered statements. If this presents a problem (such as one or more statement numbers being omitted) the following procedure will correct the problem:

a)   LIST the program (do not SAVE it) to a temporary disk file.

b)   SCRatch the User Area.

c)   ENTER the temporary disk file.

d)   RENUMBER as desired.

Examples:

>>LIST

```
    11 INPUT A
    24 INPUT B
    37 PRINT A*B
    50 GOTO 11
    63 END
```

>>RENUMBER   (default parameters are 10,10)

>>LIST

```
    10 INPUT A
    20 INPUT B
    30 PRINT A*B
    40 GOTO 10
    50 END
```

>>RENUMBER 100 (if n is not specified, n = m)

>>LIST

```
    100 INPUT A
    200 INPUT B
    300 PRINT A*B
    400 GOTO 100
    500 END
```

```
>>RENUMBER 100,10

>>LIST

  100 INPUT A
  110 INPUT B
  120 PRINT A*B
  130 GOTO 100
  140 END

>>RENUMBER 1000,150,120      (Begin renumbering
                             at line 120 in the
                             current program.)

>>LIST

   100 INPUT A
   110 INPUT B
  1000 PRINT A*B
  1150 GOTO 100
  1300 END

>>RENUMBER 1000,1,110,1150      (RENUMBER lines
                                110 through 1150
                                in the current
                                program.  The
                                new line numbers
                                will start at
                                1000 and use an
                                increment of 1.)

>>LIST

   100 INPUT A
  1000 INPUT B
  1001 PRINT A*B
  1002 GOTO 100
  1300 END
```

instruction: RUN

        format: [Ln] RUN

                  [Ln] RUN svar

where:

    Ln   is an optional line number.

    svar   is a string variable or string
              literal file reference.

The RUN instruction instructs the computer to
execute a program starting at the lowest numbered
line.

If svar is omitted, the program which is current in
the User Area is executed.

If svar is included, it must be the name of a SAVEd
program. This program will be LOADed into the User
Area and executed.

Notes:

1.   The RUN instruction, if given with a file
     reference, must reference a program which has
     been SAVEd under the version of BASIC and the
     version and size of the Cromemco Disk
     Operating System (CDOS) which is currently
     being used.

     LISTed files are compatible between different
     versions of Cromemco 16K Extended BASIC as
     well as different versions and sizes of CDOS;
     SAVEd files are not.

2.   The RUN instruction resets or clears all
     variables, string variables, and matrices.

3.   RUN sets the trigonometric mode to RADians.

4.   RUN resets ON ERROR and ON ESCAPE instructions
     to their default modes. This means that run-
     time non-fatal errors as well as the use of
     the ESCape key will cause a running program to
     abort and BASIC to display an error message.

5.   RUN sets the variable mode to that which was
     last specified. The default mode is Long

Floating Point. Refer to the Appendix, Areas
of User Interest, if it is necessary to change
the default mode.

instruction: SAVE

format: [Ln] SAVE svar

where:

Ln   is an optional line number.

svar   is a string variable or string
          literal file reference.

The SAVE instruction is used to SAVE the current
program on a disk or other file device in internal
machine format.

Note:

1.   A program which has been SAVEd on a disk file
      can be read back using the LOAD or RUN
      instructions.   A SAVEd program can only be
      LOADed or RUN with the same version of BASIC
      and the same version and size of CDOS it was
      SAVEd under.

instruction: SCRatch

format: [Ln] SCRatch

where:

Ln   is an optional line number.

The SCRatch instruction deletes the current program
from the User Area.

Notes:

1.   The programmer should keep in mind that the
      SCRatch command erases everything in the user
      work space and that SCRatched programs cannot
      be recovered.

2.   Once the work space has been cleared, the user
      may input a new program or access a SAVEd or
      LISTed program which has been stored on disk.

3.   SCRatch sets the trigonometric mode to
      RADians.

4.   SCRatch resets the variable mode to the
      default mode, normally the Long Floating Point
      Mode.

5.   SCRatch does not reset the ECHO or ESCape
      mode.

6.   SCRatch closes all open files before
      performing the actual SCRatch. If any file
      cannot be closed (because of disk or I/O
      problems) some other files may be left open
      and the User Area will not be SCRatched. The
      other files may be closed via the CLOSE\n\
      instruction.

Example:

>>LIST

  10 X=4
  20 INPUT Y
  30 Z=X*2+Y
  40 PRINT Z
  50 END

>>SCR

>>LIST

>>

In the above example, all statement lines are deleted from memory. The user can now input a new program.

instruction: TRACE

format: [Ln] TRACE

where:

Ln   is an optional line number.

The TRACE instruction sets the TRACE mode so that
the user can follow the execution of a program line
by line. When in the TRACE mode BASIC will list
the line number of each statement being executed.
Statement line numbers are listed in angle
brackets.

Example:

```
>>LIST

    10  TRACE
    20  INPUT X
    30  PRINT"THIS IS ";X
    40  LET Y=X+1
    50  PRINT Y
    60  END

>>RUN
<20>
? 10
<30>
THIS IS 10
<40>
<50>
11
<60>
***60 END***
```

instruction: No TRACE

format: [Ln] NTRACE

where:

Ln   is an optional line number.

The NTRACE instruction resets the TRACE mode so
that statement numbers are not printed during
program execution.

instruction: REM

format: [Ln] REM text

where:

Ln   is an optional line number.

text   is any string of characters.

The REM instruction is used to insert remarks or comments in a program.

Notes:

1.   REM statements included in a BASIC program are ignored when the program is executed but are output exactly as entered when the program is LISTed.

2.   REM statements occupy space in the User Area. With some long programs, or those with large lists or matrices, it may be necessary to minimize the use of REM statements in order to accommodate the program.

3.   Any grammatical or typing mistakes which are made when inputting a REM statement will not generate an error message and will be output precisely as they appear in the statement line.

4.   The programmer is encouraged to use REM statements liberally throughout a program to describe program operation. These remarks can be particularly helpful to any one who wishes to use or modify a program written by another person.

instruction: LET

    format: [Ln] LET var = exp

          or

          [Ln] var = exp

where:

    Ln   is an optional line number.

    var   is a numeric or string variable
                or a reference to an element of
                a matrix.

    exp   is the value to be assigned to
                var. It may be any expression,
                variable, constant, function,
                string variable, or literal.

The LET instruction is used to assign a value to a
given numeric variable, string variable, or element
of a matrix. The equal sign (=) is called the
assignment operator. Refer to section 6.2.

Notes:

1.   BASIC is designed to allow the user to assign
      values to variables without entering LET each
      time. This capability is called implied LET.

2.   When a string variable is used with a LET
      instruction, the portion of the string which
      is referenced is set equal to null characters
      before the source is moved into the string
      variable. The whole string is referenced if
      no subscripts follow the string variable,
      while various substrings may be referenced by
      the use of subscripts. Refer to section
      5.2.3, Referencing String Variables, for a
      complete discussion of the subject. The LET
      instruction will not move more characters than
      can be accepted by the destination string or
      substring which is being referenced.

3.   In BASIC, the equal sign is also used as a
      relational operator. Refer to section 6.3.

4.   String variables or literals may only be
      assigned to string variables while any

expression or function yielding a numeric
result may only be assigned to a numeric
variable.

5. LET and implied LET instructions execute with
    equal speed.

Examples:

| | Equivalent Implied |
LET Instructions	LET Instructions
LET A = 45	A = 45
LET B = A + 10	B = A + 10
LET A$ = "16K BASIC"	A$ = "16K BASIC"
LET X = Y + L	X = Y + L
LET G = Q(10,5)	G = Q(10,5)

instruction: MATrix

format: MAT M = aexp

where:

aexp   is an arithmetic expression,
            variable, or constant.

The MAT instruction is used to set all elements in
a matrix (M) equal to the value of the arithmetic
expression (aexp).

Notes:

1.   Matrix M must be explicitly dimensioned.

2.   This instruction is not in the instruction set
     of STAND ALONE BASIC.

Example:

```
>>LIST

    10 DIM A(4)
    20 READ A(1),A(2),A(3),A(4)
    30 DATA 20,21,22,23
    40 PRINT A(1),A(2),A(3),A(4)
    50 MAT A=0
    60 PRINT A(1),A(2),A(3),A(4)
    70 MAT A=1
    80 PRINT A(1),A(2),A(3),A(4)
    90 END

>>RUN
20            21            22            23
0             0             0             0
1             1             1             1
***90 END***
```

instruction: DEGree

format: [Ln] DEG

where:

Ln   is an optional line number.

The DEG instruction sets the trigonometric calculation mode to DEGree.

Note:

1.   RUN, SCRatch, and LOAD will automatically reset the trigonometric calculation mode to RADian.

instruction: DIM

    format: [Ln] DIM M$(aexp-1)

               [Ln] DIM M(aexp-1)

               [Ln] DIM M(aexp-1,aexp-2)

               [Ln] DIM M(aexp-1,aexp-2,aexp-3)

where:

   Ln   is an optional line number.

   M   is a numeric matrix variable.

   M$ is a string variable.

   aexpl-3 are arithmetic expressions, variables or constants.

The DIM instruction is used to define the size of a matrix or a string variable. Cromemco BASIC permits the user to define one, two, or three dimensional matrices.

Notes:

1.   A DIMensioned numeric matrix variable can have the same name as any other numeric variable. A DIMensioned string variable must have the name of a string variable. Refer to sections 5.1.1 and 5.2.1.

2.   If a matrix or string variable is not specifically dimensioned in a program, the default value of 10 (11 elements, numbered 0 through 10) will be automatically assigned to a singly subscripted matrix or string variable. Doubly and triply subscripted matrices will generate an error message if not explicitly dimensioned.

3.   The maximum size of any matrix is only restricted by the amount of available memory. Any single dimension may not exceed 16382.

4.   The dimension of a string variable may not exceed 32766.

5.   The first element in a matrix is numbered 0 (zero indexing).

instruction: IMODE

format: [Ln] IMODE

where:

Ln   is an optional line number.

The IMODE instruction changes the default mode of all variables to the Integer mode.

Notes:

1.   This instruction takes effect only after the execution of a RUN instruction after the IMODE instruction has been given. See the following example.

2.   Integer variables occupy 2 bytes and must be within the range +32767 to -32768.

3.   This instruction will be overridden by the LONG and SHORT instructions.

Example:

    1 IMODE : X=2.5 : IF X=2.5 THEN RUN

This line, appearing as the first line of a program, will ensure that the interpreter is in the integer mode. If the line is encountered while the interpreter is not in the integer mode, the IMODE instruction will be given, X will be set equal to 2.5 (a non integer number), and if X=2.5 (as will be the case if SFMODE or LFMODE are current) the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will be given after the IMODE instruction and the current mode will be integer. When the value 2.5 is assigned to X, X will be an integer variable so that 2.5 will be rounded and X will have the value of 3. Then X=2.5 will be false and control will be transferred to the next line.

instruction: INTEGER

format: [Ln] INTEGER A, B(X),...

where:

Ln   is an optional line number.

A   is a scalar variable

B   is a matrix

X   is an optional dimension

The INTEGER instruction is used to declare a given variable as INTEGER.

Notes:

1.   DIMensioning may be done via the INTEGER instruction.

2.   The INTEGER instruction must appear before the variable is referenced for the first time.

3.   Integer variables occupy 2 bytes and must be within the range +32767 to -32768.

4.   This instruction overrides the SFMODE and LFMODE instructions.

instruction: LFMODE

format: [Ln] LFMODE

where:

Ln   is an optional line number.

The LFMODE instruction is used to set all variables within a program to the Long Floating Point mode.

Notes:

1. This is the standard default mode for all variables and arithmetic operations.

2. This instruction takes effect only <u>after</u> the execution of a RUN instruction <u>after</u> the LFMODE instruction has been given. See the following example.

3. Long Floating Point variables occupy 8 bytes and must be within the range +9.99E+62 to +9.99E-65. They have an accuracy of 14 digits.

4. This instruction will be overridden by the INTEGER and SHORT instructions.

5. See the Appendix, Areas of User Interest, if it is necessary to change the default mode.

Example:

1 LFMODE : X=0.12345678 : IF X<>0.12345678 THEN RUN

This line, appearing as the first line of a program, will ensure that the interpreter is in the Long Floating Point mode. If the line is encountered while the interpreter is <u>not</u> in the Long Floating Point mode, the LFMODE instruction will be given, X will be set equal to 0.12345678 (a number which cannot be represented in either Short Floating Point or Integer modes), and if X does not equal 0.12345678 (as will be the case if SFMODE or IMODE are current) the RUN instruction will be executed. Program execution will then begin over again, this time the RUN instruction will be given <u>after</u> the LFMODE instruction and the current mode will be Long Floating Point. When the value

Ø.12345678 is assigned to X, X will be a Long
Floating Point variable with the value of
Ø.12345678. Then X<>Ø.12345678 will be false and
control will be transferred to the next line.

instruction: LONG

format: [Ln] LONG A, B(X),...

where:

Ln   is an optional line number.

A   is a scalar variable.

B   is a matrix.

X   is an optional dimension.

The LONG instruction is used to set a given variable to the Long Floating Point mode.

Notes:

1.   DIMensioning may be done via the LONG instruction.

2.   The LONG instruction must appear before the variable is referenced for the first time.

3.   Long Floating Point variables occupy 8 bytes, have an accuracy of 14 digits, and must be within the range $\pm9.99E+62$ to $\pm9.99E-65$.

4.   This instruction overrides the IMODE and SFMODE instructions.

instruction: RADian

format: [Ln] RAD

where:

Ln   is an optional line number.

The RAD instruction sets the RADian mode for trigonometric calculations.

Note:

1.   SCRatch, RUN, and LOAD will automatically reset the trigonometric mode to RADian.

instruction: SFMODE

format: [Ln] SFMODE

where:

Ln   is an optional line number.

The SFMODE instruction is used to set all variables
within a program to the Short Floating Point mode.

Notes:

1.   This instruction takes effect only <u>after</u> the
      execution of a RUN instruction <u>after</u> the
      SFMODE instruction has been given. See the
      following example.

2.   Short Floating Point variables occupy 4 bytes,
      have an accuracy of 6 digits, and must be
      within the range +9.99E+62 to +9.99E-65.

3.   This instruction will be overridden by the
      LONG and INTEGER instructions.

Example:

      1 SFMODE : X=0.90000001 : IF X<>0.9 THEN RUN

This line, appearing as the first line of a
program, will ensure that the interpreter is in the
Short Floating Point mode. If the line is
encountered while the interpreter is <u>not</u> in the
Short Floating Point mode, the SFMODE instruction
will be given, X will be set equal to 0.90000001
and if X is not equal to 0.9 (as will be the case
if IMODE or LFMODE are current) the RUN instruction
will be executed. Program execution will then
begin over again, this time the RUN instruction
will have been given <u>after</u> the SFMODE instruction
and the current mode <u>will</u> be Short Floating Point.
When the value 0.90000001 is assigned to X, X will
be a Short Floating Point variable so that
0.90000001 will be rounded and X will have the
value of 0.9.   Then X<>0.9 will be false and
control will be transferred to the next line.

121

instruction: SHORT

format: [Ln] SHORT A, B(X),...

where:

Ln   is an optional line number.

A   is a scalar variable.

B   is a matrix.

X   is an optional dimension.

The SHORT instruction is used to set a given variable to the Short Floating Point mode.

Notes:

1.   DIMensioning may be done via the SHORT instruction.

2.   The SHORT instruction must appear before the variable is referenced for the first time.

3.   Short Floating Point variables occupy 4 bytes, have an accuracy of 6 digits, and must be within the range ±9.99+62 to ±9.99E-65.

4.   This instruction overrides the IMODE and LFMODE instructions.

command: CONtinue

format: CON

The CONtinue command CONtinues program execution
after a program is interrupted by a STOP statement,
a program error, or pressing the ESCape key.

Notes:

1.   Program execution will commence at the line
      following the statement at which the program
      stopped.

2.   If program execution stopped because of a
      program error, the error can be corrected and
      the CON command used to continue execution
      from the line following the one where the
      error occurred.

3.   If the user wishes to re-execute the line in
      error, GOTO (as a command) should be used.

statement: END

   format: Ln END

where:

   Ln   is a line number.

The END statement halts program execution and causes BASIC to return to the command mode.

Notes:

1.   Unlike the STOP statement, program execution may not be CONtinued after an END statement has been executed.

2.   Upon execution of the END statement, BASIC displays a message on the console indicating the line number of the END statement which caused the program to halt.

Example:

   >>LIST

     100 PRINT "Tomorrow and tomorrow and tomorrow..."
     200 END

   >>RUN
   Tomorrow and tomorrow and tomorrow...
   200 END
   >>

instruction: FOR...NEXT

format: [Ln] FOR avar=aexpl TO aexp2 [STEP aexp3]
.
.
[program instructions]
.
.
[Ln] NEXT avar

where:

Ln   are optional line numbers.

avar   is a non-subscripted numeric
            variable.   This   is   the   index
            variable.

aexpl-3   are arithmetic expressions,
            variables, or constants.   They
            may not be string literals or
            string variables.   These are the
            FOR...NEXT loop parameters.

            aexp-1 is the initial value.
            aexp-2 is the final value.
            aexp-3 is the step value.

The FOR...NEXT instructions are used to repeat a
part of a BASIC program a number of times.   During
the execution of a FOR...NEXT loop, an index (X) is
maintained.   When this index becomes equal to (or
greater than, or less than) the final value
(aexp-3), control is transferred to the instruction
following the NEXT instruction.

Notes:

1.   The following sequence defines the execution
      of a FOR...NEXT loop:

      A.   The expressions aexp-1, aexp-2, and aexp-
            3 are evaluated.   If aexp-3 is omitted it
            is given a value of +1 (if the STEP value
            is not specified it is assumed to be +1).

      B.   The index variable (avar) is set equal to
            the initial value (aexp-1).

      C.   The instructions following the FOR and
            preceding the NEXT are executed.

D.   The step value is added to the index variable (avar = avar + aexp-3).

E.   If the STEP value (aexp-3) is positive and the index variable (avar) is equal to or greater than the final value (aexp-2) then the condition for termination of the FOR...NEXT loop has been met and control is transferred to the instruction following the corresponding NEXT instruction.

If the STEP value (aexp-3) is negative and the index variable (avar) is equal to or less than the final value (aexp-2) then the condition for termination of the FOR...NEXT loop has been met and control is transferred to the instruction following the corresponding NEXT instruction.

Otherwise execution of the loop continues with C above.

2.   The FOR instruction must be the last instruction on a line. It may not be followed by another instruction on the same line.

3.   The STEP (aexp-3) portion of the FOR instruction is optional. If a STEP value is not specified, BASIC assumes a value of +1 for aexp-3.

4.   BASIC programs will execute significantly faster if the loop parameter variables are declared as type INTEGER.

5.   FOR-NEXT loops may be nested within a program. It is important to keep in mind, however, that each FOR instruction and its corresponding NEXT instruction must be completely contained within any larger loop.

6.   Program LISTings have FOR...NEXT loops indented for clarity.

Examples:

>>LIST

```
10  FOR A = 0 TO 10 STEP 2
20  B = A+1
30  PRINT B;
40  NEXT A
50 END
```

>>RUN
1357911***50END***
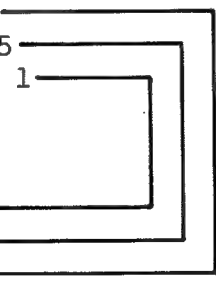
In the above example, upon entering the FOR...NEXT loop defined by lines 10 through 40, the loop parameters are evaluated and set equal to:

```
 0 (aexp-1, the initial value)
10 (aexp-2, the final value)
 2 (aexp-3, the STEP value)
```

The index variable is then set equal to the initial value (A=0). Execution continues with lines 20 and 30 where the variable A maintains the value assigned to it by the FOR instruction. At line 40 the index variable is tested to see if it is equal to or greater than the final value (is A $\geq$ aexp-2?). In this case 0 is not equal to or greater than 10, so execution continues with line 10. The STEP value is added to the index variable (A = A + aexp-3). This continues until (A $\geq$ aexp-2). Then control is transferred to statement 50.

Correct Nesting Format:

```
 10  FOR A = 1 TO 50 STEP 2
 20    FOR B = 1 TO 30 STEP 5
 30      FOR C = 1 TO 10 STEP 1
              .
              .
              .
100      NEXT C
110    NEXT B
120  NEXT A
```
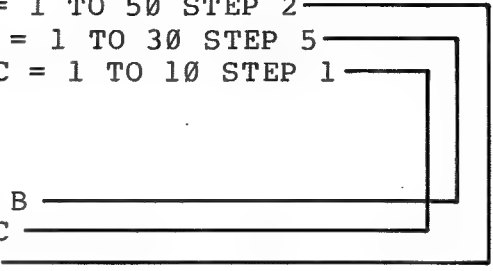
Illegal nesting occurs when FOR...NEXT loops overlap. The following example will generate a run time error.

Incorrect Nesting Format

```
10  FOR A = 1 TO 50 STEP 2
20   FOR B = 1 TO 30 STEP 5
30    FOR C = 1 TO 10 STEP 1
           .
           .
           .
100    NEXT B
110   NEXT C
120 NEXT A
```

instruction: GOSUB...RETURN

format: [Ln] GOSUB n
.
.
n [program instructions]
.
.
[Ln] RETURN

where:

Ln   are optional line numbers.

n   is the line number of the first
            statement of the subroutine to
            which control is transferred.

The GOSUB instruction transfers control to a
subroutine.

When, during the execution of the subroutine, a
RETURN instruction is executed, control is passed
to the instruction following the GOSUB instruction
which called the subroutine.

Notes:

1.   In BASIC, subroutines may be fully enclosed or
      nested in other subroutines. Subroutines may
      be nested up to 16 deep.

      When nesting subroutines, remember that the
      RETURN instruction will take you back to the
      last GOSUB and start execution of the
      instruction immediately following. Improperly
      nested GOSUB...RETURN instructions will
      generate runtime error messages. A nested
      subroutine is executed after the GOSUB
      statement and before the RETURN statement of
      the subroutine in which it is enclosed.

2.   The GOSUB and the RETURN instruction must each
      be the last instruction on a line. They may
      not be followed by another instruction on the
      same line.

Example:

```
>>LIST

    20 GOSUB 50
    30 PRINT "PROGRAM OVER"
    40 END
    50 PRINT "THIS IS A SUBROUTINE"
    60 PRINT "WHICH DEMONSTRATES THE"
    70 PRINT "GOSUB STATEMENT"
    80 PRINT
    90 RETURN

>>RUN
THIS IS A SUBROUTINE
WHICH DEMONSTRATES THE
GOSUB STATEMENT

PROGRAM OVER
***40 END***
```

instruction: GOTO

format: [Ln] GOTO n

where:

Ln   is an optional line number.

n   is the line number of the
          statement to which control is
          transferred.

The GOTO instruction unconditionally transfers
control to the statement line specified by n.

When used as a statement, GOTO interrupts the
normal execution sequence of program statements and
transfers control to the specified statement.

When used as a command GOTO will cause execution of
a program to start with the specified statement.

Notes:

1.   When used with the IF...THEN instruction, the
      words GOTO are optional:

          IF X=0 THEN GOTO 50

             or

          IF X=0 THEN 50

      are equivalent and are both legal
      instructions.

2.   When execution of a program is initiated or
      continued using a GOTO command, no variable
      initialization takes place.

Example:

>>LIST

```
10  INPUT A
20  IF A<0 THEN 200
30  B = SQR(A)
40  PRINT "THE SQUARE ROOT OF ";A;" IS ";B
50  GOTO 210
200 PRINT "THIS YIELDS AN IMAGINARY NUMBER"
210 END
```

```
>>RUN
? - 2
THIS YIELDS AN IMAGINARY NUMBER
***210 END***
>>RUN
? 9
THE SQUARE ROOT OF 9 IS 3
***210 END***
```

In statement 20 the words GOTO are omitted. This
statement causes control to be transferred to
statement 200 if the condition (A<0) is true.
Statement 50 unconditionally transfers control to
statement 210.

instruction: IF...THEN

format-1:   [Ln] IF exp THEN n

format-2:   [Ln] IF exp THEN instruction

[Ln] IF exp THEN instruction:instruction:...

where:

Ln   is an optional line number.

exp   is a relational or arithmetic
            expression, an arithmetic
            variable, or a constant.

n   is the line number of the
            statement to which control is
            transferred.

instruction is any BASIC instruction except
             FOR, NEXT, END, REM or DATA.

The IF...THEN instruction evaluates exp to false
(=Ø) or true (not equal to Ø).

If format-1 is used, control is transferred to the
specified statement if exp is evaluated as true,
and to the next sequential statement if exp is
false.

If format-2 is used, and exp is true, the
instructions remaining on the same line are
executed before control is transferred to the next
sequential statement. If exp is false, control
passes to the next sequential statement line.

Notes:

1.   In a relational expression, a relational
     operator (=, <, >=, <=, <>, or >) is used to
     compare two expressions or values. Refer to
     sections 6.3 and 6.4 for a discussion of
     relational and boolean operators.

2.   No instruction may follow format-1 of the
     IF...THEN instruction on the same line.

3.   Standard BASIC format must be followed for all
     instructions used with format-2. This
     includes the restriction that no instruction

may follow a FOR, GOTO or GOSUB instruction on
the same line.

4.   IF...THEN instructions may be nested using
format-2. For example:

        100 IF A = B THEN IF G$ = "N" THEN 500

is a legal statement.

Examples:

        100 INPUT A
        110 IF A=0 THEN @"MUST BE NON-ZERO": GOTO 100
        120 PRINT A
        130 END

In this example, the computer outputs a prompt (?)
to which the user responds with a number. If the
number is non-zero (A=0 is false), control will be
passed to line 110, the number will be printed, and
execution of the program will terminate. If the
number is zero (A=0 is true), the part of line 100
after THEN will be executed, printing out the
message and returning control to line 100 which
will request another number from the user.

        100 INPUT "First Name: ", A$
        200 IF A$ = "FRED" OR A$ = "Fred" THEN GOTO 400
        300 GOTO 100
        400 PRINT "So you are FRED!"
        500 END

This program will continue to prompt the user with
"First Name:" until the user responds with either
Fred or FRED. The IF...THEN instruction includes a
compound relational expression:

    A$ = "FRED" OR A$ ="Fred"

This expression is evaluated as true (=1) if either
FRED (all upper case) OR Fred (Upper case F, lower
case red) is entered. This type of checking is
very useful in interactive programs where a variety
of user responses are to be allowed.

instructions: ON...GOTO
               ON...GOSUB

     format: [Ln] ON (aexp) GOTO n1,n2,...,ni

               [Ln] ON (aexp) GOSUB n1,n2,...,ni

where:

   Ln   is an optional line number

   aexp   is an arithmetic expression,
              variable, or constant

The ON...GOTO and ON...GOSUB instructions transfer
control to any one of several lines in a program
based on the value of the expression (aexp)
contained in the instruction.

Refer to the GOTO and GOSUB instructions for more
information.

Notes:

1. When aexp is evaluated, if it is equal to one,
    control will be passed to the statement
    numbered n1; if it is equal to two, control
    will be passed to the statement numbered n2;
    if it is equal to i, control will be passed to
    the statement numbered ni.

2. If aexp evaluates to a non integer number the
    value of the number will be rounded to the
    nearest integer for the purpose of these
    instructions.

3. If aexp is evaluated as less than one or
    greater than i the instruction will be ignored
    and control will pass to the next sequential
    instruction.

4. If n1, n2, or ni is a nonexistant line number,
    and if control is transferred to that line, a
    fatal run time error will be generated.

5. No instruction may follow an ON...GOTO or
    ON...GOSUB instruction on the same line.

statement: STOP

format: Ln STOP

where:

Ln   is a line number.

The STOP statement halts program execution and
causes BASIC to return to the command mode.

Notes:

1.   After a program has been STOPed by a STOP
     statement execution may be restarted from the
     statement line immediately following the line
     containing the STOP statement by the use of
     the CONtinue command.

2.   Upon execution of the STOP statement, BASIC
     displays a message on the console indicating
     the line number of the STOP statement which
     caused the program to halt.

Example:

```
>>LIST

    10  INPUT A,B,C,D,E,F
    20  LET N=A+B/C
    30  PRINT A;B;C
    40  STOP
    50  PRINT D;E;F
    60  PRINT N
    70  END

>>RUN
? 1,2,3,4,5,6
123
***40 STOP***
>>CON
456
1.6666666666667
***70 END***
```

In this example, the BASIC returns to command mode
after encountering the STOP in statement line 40.
The program is then CONtinued when the user enters
the CON command.

instruction: INPUT (from the console)

   format: [Ln] INPUT X1,X2,...Xn

       [Ln] INPUT "string", X1,X2,...Xn

where:

  Ln   is an optional line number.

  X1...Xn are numeric or string variables.

  string is an optional string literal.
       It is a prompt to be displayed
       on the console upon encountering
       the INPUT instruction.

The INPUT instruction assigns a value, input from
the console, to a variable. When the INPUT
instruction is executed, a prompt (either a
question mark (?) or a string as used above) is
output to the console. The user should respond to
this prompt by entering a list of data which
corresponds to the variable list in the INPUT
instruction.

Notes:

1. If the string format of the INPUT instruction
  is used, the string will replace the question
  mark as the initial prompt.

2. If the user types in fewer data items than are
  called for in the variable list, a double
  question mark will appear on the terminal.
  This prompt continues to appear until each
  variable has been assigned a value. See
  examples.

3. If more data is input than is required, an
  error message will be generated.

4. The type of data input must be the same as the
  type of variable listed in the INPUT command.
  For example, if the INPUT command contains a
  list of numeric variables, the data input must
  be numeric data. If an attempt is made to
  INPUT string data into a numeric variable, an
  error message will be generated.

5. When a string variable is used with an INPUT

instruction, the portion of the string which
is referenced is set equal to null characters
before the source is moved into the string
variable. The whole string is referenced if
no subscripts follow the string variable,
while various substrings may be referenced by
the use of subscripts. Refer to section
5.2.3, Referencing String Variables, for a
complete discussion of the subject. The INPUT
instruction will not move more characters than
can be accepted by the destination string (or
substring) which is being referenced.

6.   If the INPUT list is terminated with a
semicolon, the carriage RETURN which the user
types after the requested data has been
entered will not be echoed. This allows more
than one prompt and response sequence to
appear on a single line.

7.   Refer also to the description of the INPUT
instruction in the chapter on Data File I/O.

Examples:

```
>>LIST
    10 INPUT A,B,C,D
    20 PRINT A
    30 INPUT E,F
    40 PRINT B;" ";C;" ";D;" ";E;" ";F
    50 END

>>RUN
? 32,18,20,4
32
? 100,200
18 20 4 100 200
***50 END***
```

In the above example, the proper number of items
was INPUT in response to each prompt. Suppose we
were to respond to INPUT statement 10 with only
three numbers:

```
>>RUN
? 5,10,15
?? 20
5
? 30,40
10 15 20 30 40
***50 END***
```

BASIC displayed an additional prompt (??) when it did not receive as many items as were in the INPUT list.

In our example, line 10 and line 30 can be replaced with lines which will tell the user specifically what items are needed:

```
>>10 INPUT "Enter four numbers:  ",A,B,C,D
>>30 INPUT "The last two numbers? ",E,F
>>RUN
Enter four numbers:  100,200,300,400
100
The last two numbers?  500,600
200 300 400 500 600
***50 END***
```

instruction: PRINT (to the console)

    format: [Ln] PRINT

            [Ln] PRINT A1,A2,...,An

where:

Ln   is an optional line number.

A1-n   is a number, string literal
           enclosed in quotation marks, a
           numeric variable, a string
           variable, a numeric expression,
           or a standard or user defined
           function.

The PRINT instruction is used to output information
to the terminal.

Notes:

1.   When used alone, the PRINT instruction will
      generate a linefeed.

2.   The at sign (@) may be used in place of the
      word PRINT for brevity.

      Example:

          30 PRINT
          40 PRINT "AND IT SAVES SPACE"

      can also be written as:

          30 @
          40 @ "AND IT SAVES SPACE"

      This facilitates entry of large, text-oriented
      programs with many blank lines.

3.   The spacing of output can be controlled by
      using a comma (,) or a semicolon (;) between
      items in a PRINT instruction list.

      Items separated by commas are printed
      beginning in the leftmost column of each
      printing field. Using the default value of 20
      columns per field, the PRINT statement below:

        10 PRINT 1,2,3,4

will produce output in the following format:

             1   2   3   4

column   Ø   2Ø   4Ø   6Ø
    no.

See the SET instruction to change the default
value of the number of columns per field.

If a semicolon is used between items, the
items are printed adjacent to each other
(i.e., without spaces between items). For
example, the statement:

    10 PRINT "AL"; "TO"; "GE"; "TH"; "ER"

    >>RUN

will produce output in the following format:

    ALTOGETHER

Commas and semicolons may be used in a PRINT
statement in any combination.

4.   If more items are listed in a PRINT statement
     than can be output on one line, BASIC will
     generate a linefeed and continue printing on
     the next line.

5.   More complex formatting of printed text is
     possible with the TAB and SPC functions and
     the PRINT USING instruction.

6.   Refer also to the description of the PRINT
     instruction in the chapter on Data File I/O.

Example:

```
>>LIST

    10 A=1979
    20 REM B=YOUR BIRTHDATE
    30 PRINT "THIS IS ";A
    40 PRINT
    50 PRINT
    60 INPUT B
    70 C=A-B
    80 PRINT "YOU WERE BORN IN ";B
    90 PRINT
   100 PRINT "YOU ARE "; C;" YEARS OLD THIS YEAR"
   110 END

>>RUN
THIS IS 1979


? 1952
YOU WERE BORN IN 1952

YOU ARE 27 YEARS OLD THIS YEAR
***110 END***
```

instruction: READ

format: [Ln] READ A1,A2,...,An

where:

Ln   is an optional line number.

A1-n   are numeric or string variables.

The READ instruction is used to read values from
DATA statement(s) and assign these values to the
READ instruction variable list.

Notes:

1.   The order in which variables appear in the
      READ instruction determines which value from
      the DATA list will be assigned to which
      variable. For instance, the first value that
      appears in the DATA list is assigned to the
      first variable in the READ list, the fifth
      value is assigned to the fifth variable, and
      so forth.

2.   A pointer is moved in sequence through the
      list of DATA values as these values are
      assigned to variables in the READ list. The
      number of DATA elements must be equal to or
      greater than the number of variables in the
      READ list. If there are fewer items remaining
      in the DATA list than are in the current READ
      list, an error message will be generated.

3.   DATA elements corresponding to numeric
      variables must be numeric data and elements
      corresponding to string variables must be
      string literals.

4.   When a string variable is used with a READ
      instruction, the portion of the string which
      is referenced is set equal to null characters
      before the source is moved into the string
      variable. The whole string is referenced if
      no subscripts follow the string variable,
      while various substrings may be referenced by
      the use of subscripts. Refer to section
      5.2.3, Referencing String Variables, for a
      complete discussion of the subject. The READ
      instruction will not move more characters than
      can be accepted by the destination string (or

substring) which is being referenced.

Example:

```
>>LIST

     10 READ A,B,C,D$
     20 PRINT A,B,C,D$
     30 DATA 10,20,30,"END"
     40 END

>>RUN
10              20              30              END
***40 END***
```

instruction: RESTORE

format: [Ln] RESTORE

[Ln] RESTORE n

where:

Ln   is an optional line number.

n   is a line number in the current
             program. This must be a line
             number of or before a DATA
             statement. If n is not used,
             the DATA list pointer is
             positioned before the first DATA
             list item in the program. If n
             is included, the DATA list
             pointer is positioned before the
             first occurring DATA list item
             after the specified line number.

The RESTORE instruction resets the DATA list
pointer which is associated with the READ
instruction. This allows the user to reread or
skip over DATA items.

Example:

>>LIST

```
 10 READ A,B,C,D
 20 READ E,F,G,H
 30 RESTORE
 40 READ R,S,T,U
 50 RESTORE 90
 60 READ L,M,N,O
 70 DATA 1,2,3,4
 80 DATA 5,6,7,8
 90 DATA 9,10,11,12
100 PRINT A,B,C,D
110 PRINT E,F,G,H
120 PRINT R,S,T,U
130 PRINT L,M,N,O
```

>>RUN

1	2	3	4
5	6	7	8
1	2	3	4
9	10	11	12

END

statement: DATA

format: Ln DATA Al, A2,...,An

where:

Ln   is a line number.

Al-An   are   numeric   expressions,
            constants, or string literals.

The DATA statement specifies values for variables
appearing in a READ instruction.

Notes:

1.   The READ instruction is used to read values
     from DATA and assign these values to the READ
     instruction variable list.

2.   The RESTORE instruction allows items in the
     DATA list to be skipped or reread by resetting
     the DATA list pointer.

Example:

10 DATA 7*3,9+(2*107),"This is DATA",15,75

instruction: PRINT USING

format: [Ln] PRINT USING svar, X1, X2,...,Xn

where:

Ln   is an optional line number.

svar   is a string literal or string
          variable format definition.

X1-n   are any numeric, string or
          matrix variables, string
          literals, or constants.

The PRINT USING instruction allows the user to
specify a format for printing output.

Notes:

In examples in this section the character b
represents a blank character.

1.   A format field is bounded on either side by
      any character which is not one of the special
      characters (# & * + , $! - .). As indicated
      in the general format for the PRINT USING
      instruction, a format expression may include
      more than one format field and may also
      include string literals. If multiple format
      fields are specified, the values of
      expressions are assigned to these fields in
      order. A format expression may also be
      assigned to a string variable.

      Example:

          >>LIST

              10 A=11 : B=333 : C=22
              20 DIM D$(20) : D$="### &&&&M&&&"
              30 PRINT USING"### &&&&M&&&",A,B,C
              40 @USING D$,A,B,C
              50 END

          >>RUN
           11 0333M022
           11 0333M022
          50 END

      Statement 30 in the above example outputs the

147

three variables (A, B, and C) with 2 spaces
separating the first two and the literal M
separating the last two. Any literal which is
not one of the PRINT USING special characters
may be used to separate format fields. This
literal will be printed between the fields if
it is a blank or any other character.
Statement 40 outputs the same information in
the same format using a string variable
instead of a string literal in the PRINT USING
instruction. Statement 40 also abbreviates
PRINT with the symbol @.

2.   When a string literal appears in a format
expression, the characters of the string
literal are printed in the positions held by
any of the special format field characters.
Strings are left justified in the format
field. If the number of characters in the
string literal is less than the number of
characters in the format field, the extra
spaces will be blank filled. If the number of
characters in the string literal is greater
than the number of characters in the format
field, the extra characters in the string
literal will be truncated.

Examples:

In the instruction:
      PRINT USING "****,**.*", "STRING"
the string literal is output in the format:
      STRINGbbb

In the instruction:
      PRINT USING "&&&,&&&.&&","STRING-LITERAL"
the string literal is output in the format:
      STRING-LIT

3.   If the number of items in the expression list
exceeds the number of specified format fields,
the specified format fields will be re-used
for the extra items.

Examples:

In the instruction:
PRINT USING "$$&&.&&", A,B,C
the expressions A, B, and C will all be
PRINTed with the format field $$&&.&&.

148

In the instruction:
PRINT USING "$$##bb$$&&.&", A,B,C
the expressions A and C will be formatted
using the format field $$## and the expression
B will be formatted using the format field
$$&&.&.

4. All normal PRINT functions (such as TAB, SPC,
    semicolon, and comma) are overridden by the
    PRINT USING instruction.

5. The format expression may include a maximum of
    128 characters.

6. If a number being formatted has more digits
    than allowed for in the format expression, an
    all asterisk error message will result.

Digit Formatting

These special characters may be used to format
digits:

    #   indicates leading blanks
    &   indicates leading zeroes
    *   indicates leading asterisks

These symbols are used to right justify digits in a
print field. The width of this print field is
determined by the number of special characters
included in a format field. Any non-digits (such
as a minus sign) are eliminated. If the number of
special characters in the format field exceeds the
number of digits in the expression, the digits will
be right justified within the field and preceded by
characters corresponding to the special characters
used in the format field. In the following
examples, the character b is used to represent
blank characters (spaces).

Examples:

Value of Expression	Format Field	Output
1	#####	bbbb1
12	#####	bbb12
123	#####	bb123
1234	#####	b1234
12345	#####	12345
123456	#####	*****

```
1               &&&&&           00001
12              &&&&&           00012
123             &&&&&           00123
1234            &&&&&           01234
12345           &&&&&           12345
123456          &&&&&           *****

1               *****           ****1
12              *****           ***12
123             *****           **123
1234            *****           *1234
12345           *****           12345
123456          *****           *****
```

Comma (,)

The comma (,) places a comma in the position in
which it appears in the format field. If the
format specifies that a comma be output in a
position in the field which consists of leading
blanks, zeroes, or asterisks, then a blank, a zero,
or an asterisk respectively are printed in the
comma position.

Examples:

Value of Expression	Format Field	Output
2003	##,###	b2,003
4	##,###	bbbbb4
4457	&&&,&&	044,57
18	&&&,&&	000018
996546	*****,*	99654,6
22	*****,*	****2,2

Decimal Point (.)

The decimal point (.) places a decimal point in
the position in which it appears in the format
field. All digit positions which follow the
decimal point are filled with digits. If the
expression contains fewer fractional digits than
are specified, zeroes will be printed in the extra
positions. If the expression contains more
fractional digits than are specified, the
expression will be rounded so that the number of
fractional digits equals the number of format

150

positions available.

Examples:

Value of Expression	Format Field	Output
234	###.###	234.000
23.4567	###.###	b23.457
13	&&.&	13.0
66.72319	&&.&	66.7
876.1245	*****.**	**876.12
1234567.245	*****.**	*******

In the last example, when too many significant
digits appear to the left of a decimal point, an
all asterisk error message results.

Fixed Plus (+) and Minus (-) Signs

The plus (+) and minus (-) signs may appear in the
first character position in a format field. The
character + or - will print the respective sign of
an expression in the specified character position
in the format field. When an expression is
preceded by a plus or minus sign, any leading
zeroes will be replaced by blanks, zeroes, or
asterisks as specified. When a positive expression
is preceded by a minus sign, a blank space is left
in the sign position.

Examples:

Value of Expression	Format Field	Output
56.8888	+##.###	+56.889
4.564	+##.###	+b4.564
6.456	+&&&.&&	+006.46
234.2	+&&&.&&	+234.20
-23.56	-*****.*	-***23.6
2345	-*****.*	*2345.0
-2345678.34	-*****.*	*******

Floating Plus (++) or Minus (--) Signs

The use of two or more plus or minus signs at the
beginning of a format field will output the
respective sign directly preceding the value of the
expression. If a positive expression is PRINTed
USING a floating minus format, a blank is printed
immediately preceding the number instead of a minus
sign. The additional signs in the floating point
format can be used to represent digits.

Examples:

Value of Expression	Format Field	Output
2.234	++##.###	b+b2.234
22.234	++##.###	b+22.234
-44.56	--&&&.&	b-044.6
5.32	--&&&.&	bb005.3
178.456	++****.**	b+*178.46
12345678.45	++****.**	********
55.17	+++++.++	bb+55.17
55.17	-----.--	bbb55.17
-55.17	-----.--	bb-55.17

Fixed Dollar Sign ($)

The dollar sign ($) is used in either the first or
second character position in the format field to
print out a dollar sign in that position. A dollar
sign specified in the second position of the format
field must be preceded by either a plus (+) or a
minus (-) sign.

Examples:

Value of Expression	Format Field	Output
23.456	$##.##	$23.46
4.52	$##.##	$b4.52
-57.654	-$&&&.&&&	-$057.654
123.7789	-$&&&.&&&	b$123.779
2.34	$*.*	$2.3
234.55	$*.*	****

Floating Dollar Sign ($$)

The use of two or more dollar signs beginning at
either the first or second character position in
the format field will output a dollar sign
immediately preceding the value of an expression.
If two or more dollar signs begin in the second
character position, the first character position
must contain a plus or a minus sign.

Examples:

Value of Expression	Format Field	Output
234.2345	$$$$#.###	b$234.235
4.45	$$$$#.###	bbb$4.45Ø
23.989	$$&&.&&	b$23.99
4.5	$$&&.&&	b$Ø4.5Ø
24.56	-$$***.**	bb$*24.56
-4455.67	-$$***.**	-$4455.67

Exponent Fields (!!!!)

Four consecutive exclamation characters (!!!!)
indicate an exponent in the format field. These
four exclamation points represent the expression
E+nn, where n is any digit. When used with a
numeric expression, this format field will output
the expression in exponential form.

Examples:

Value of Expression	Format Field	Output
23.3456	##.##!!!!	23.35E+ØØ
2ØØØ	##.###!!!!	2Ø.ØØE+Ø2
-.36	-&&&.&&!!!!	-36Ø.ØØE-Ø3

7. Only the characters # or & should be used to
    the right of a decimal point. The asterisk
    (*) character follows the same rules as the #
    character when used to the right of a decimal
    point.

8. Either the floating dollar ($$) character or
    pluses (++) or minuses (--) may be used within
    the same formatting statement, but not both

types of characters.

9. A non-floating print character cannot be placed left of a floating character. For example, the format fields $+++ or +$$$ are legal but the format field +$++ is illegal.

10. 16K BASIC does not check comma syntax.

11. Only one decimal point may be used in a format field.

12. Trailing + or - signs are illegal in exponential format (i.e., E+nn.)

A typical application for the PRINT USING format
follows. This application is a program for
computing a payroll on an automated basis.

```
10 PRINT "ENTER SOCIAL SECURITY NUMBER AND HOURS WORKED"
20 PRINT "OF FOUR EMPLOYEES"
30 PRINT
40 INPUT A,A1
50 INPUT B,B1
60 INPUT C,C1
70 INPUT D,D1
90 PRINT "WHAT IS THE HOURLY WAGE";
100 INPUT W
110 PRINT
120 PRINT
130 A1=A1*W :  B1=B1*W :  C1=C1*W :  D1=D1*W
140 PRINT USING " ########",A,B,C,D
150 PRINT
160 PRINT USING "   $$$#.##",A1,B1,C1,D1
170 PRINT
180 PRINT "ANOTHER FOUR?  TYPE 1 IF YES, TYPE 0 IF NO"
190 INPUT Q
200 IF Q=1 THEN GOTO 10
210 END

>>RUN
ENTER SOCIAL SECURITY NUMBER AND HOURS WORKED
OF FOUR EMPLOYEES

? 552966937,40
? 525449121,40
? 455238541,35
? 211329494,32
WHAT IS THE HOURLY WAGE? 4.00

 552966937 525449121 455238541 211329494

   $160.00   $160.00   $140.00   $128.00

ANOTHER FOUR?  TYPE 1 IF YES, TYPE 0 IF NO
? 0
***210 END***
```

function: SPaCe

format: SPC (aexp)

where:

aexp   is an arithmetic expression,
            variable, or constant.

The SPC function instructs BASIC to PRINT the
specified number of spaces.

Notes:

1.   The SPC function may be used only with a PRINT
      instruction.

2.   Unlike the TAB function, which always
      determines print position relative to column
      Ø, the SPC function determines print position
      relative to the current column location.

Example:

        >>LIST

          10 A=2
          20 INPUT C,D
          30 PRINT SPC(1Ø);A;SPC(A*1Ø);C;D
          40 END

        >>RUN
        ? 2Ø,3Ø
                  2   2Ø3Ø

column no.   1Ø   31

In this example, the computer is instructed to skip
1Ø spaces, print A, skip 2Ø spaces, and print C and
D.

function: TAB

format: TAB (aexp)

where:

aexp   is an arithmetic expression,
            variable, or constant.

The TAB function causes output to begin in a
specified column.

Notes:

1.   The TAB function may be used only with a PRINT
     instruction.

2.   Multiple TAB functions may be included in one
     PRINT instruction, but the user should keep in
     mind that the print position indicated by
     successive TAB functions is always determined
     relative to column 0.

3.   Columns are numbered 0 through the page width
     so the first column is column 0.

4.   If the argument to the TAB function exceeds
     the current page width, it is reduced modulo
     that page width to a number between 0 and the
     page width. The default value for the page
     width is 79.

5.   If the argument is negative, no tabbing takes
     place.

6.   If the (reduced) argument is greater than the
     current column position, a new line (CR,LF) is
     issued and the TAB is executed on the next
     line.

Example:

        >>LIST

            10 A=1
            20 INPUT B,C,D
            30 PRINT TAB(2);B;TAB(5);B+C+D;TAB(A+9);D
            40 END

        >>RUN
        ? 5,8,9
          5 22   9
        40 END

In this example, the computer is instructed to
begin printing B in column 2, to begin printing
B+C+D in column 5, and to begin printing D in
column 10.

DATA FILE INPUT/OUTPUT

15.1   Data Files

Data is information. A file is a place and method
for storing many individual items of information.

A computer data file (or file) is defined by:

1.   The storage medium (paper tape, floppy disk,
      etc.),

2.   The method of accessing the data (sequential
      or random),

3.   The code by which the data is translated for
      storage (ASCII or internal machine
      representation).

15.1.1 Records

A record is a group of related items. A data file
is made up of records. Information is inserted
into or retrieved from the file record by record.

If a file contained information on all of the
baseball games in one year, each record might
contain information on one game, one player, or one
team. All of the records would contain similar
information. If the first record (record number 0)
held the statistics on game #1, the first number in
the first record could be the number of hits team A
got, while the second number would be the number of
hits team B got. The third number could be the
number of errors made by team A, while the fourth
number would be the number of errors made by team
B. The second record (record number 1) would
contain similar information covering game #2.

15.1.2   Fields

A field describes one item in a record. A field
can contain string (alphabetic) or numeric
information.

In the above example, instead of saying that the
first number was the number of hits team A got, we
could say that the first field was numeric and
contained the number of hits team A got. We can
now further describe the record layout by saying
that the fifth and sixth fields are alphabetic and
contain the team names.

15.2   Floppy Diskette

A floppy diskette (or disk) is a flat, round
magnetic storage medium. It is protected by a
square envelope which exposes only a small portion
of the surface of the disk in addition to the hole
in the middle. When used in conjunction with a
disk drive and disk controller, the disk becomes a
data file storage device. Data can be added to the
data already on the disk or records on the disk can
be changed or deleted.

15.3   Creating a Data File

Before the Cromemco Disk Operating System (CDOS -
the operating system through which all disk
operations are performed, even if the programmer is
using BASIC) can OPEN a disk file, it must CREATE
the file in the file directory.

A file can be created from the CDOS monitor by
using the CDOS TEXT EDITOR. A file can also be
created from BASIC by using the CREATE instruction.

15.3.1 OPENing an I/O Channel

Once a file has been CREATEd, it must be OPENed to
allow BASIC access to the file. The OPEN
instruction reserves an input/output channel and
assigns the I/O channel number to the file
reference or file device. Once a file has been
opened (and assigned a channel number) all input
and output from/to that file will be done through
the assigned channel.

BASIC normally carries 4 I/O channels in addition
to the console. One I/O channel is needed for each
file which is OPENed at the same time. Each
channel occupies 192 bytes of memory. Increasing
the number of channels allows more files to be
opened simultaneously but reduces the amount of
memory available to the BASIC user. Conversely,
decreasing the number of I/O channels increases the
amount of available memory. If you wish to change
the number of channels available for use, see the
Appendix, Changing the Number of I/O Channels.

15.3.2 CLOSE

The CLOSE instruction will disassociate the
input/output channel number and the file which were
associated by the OPEN instruction. The channel
will then be available for use by other files.

15.4 Internal Machine vs. ASCII Representation

BASIC can store data in two formats. Both consist
of groups of ones and zeros (binary
representation), but each needs to be translated
differently for output to a terminal or printer.

ASCII (American Standard Code for Information
Interchange) is the closest to written letters and
numbers. ASCII is stored in the machine according
to the table of ASCII codes (see Appendix B). Each
character code occupies one byte (8 bits). When
the code is translated from binary to decimal, the
character which is represented can be found in the
ASCII table. This is the way all BASIC strings are
stored. ASCII information is stored so that one
character is stored in one byte.

161

The need arises for another method of storing
numbers when speed of arithmetic computation and
amount of storage space become important factors.

Internal Machine code is used only for the storage
of numbers. The number of bytes occupied by a
number depend on its magnitude and precision (the
size of the number and the number of decimal places
which need to be kept). There are three internal
machine formats:

Integer   size: 2 bytes
                                 range: +32767 < N < -32768
                                 accuracy: nearest integer

Short Floating Point   size: 4 bytes
                                 range: +9.99E+62 < N < +9.99E-65
                                 accuracy: 6 decimal digits

Long Floating Point   size: 8 bytes
                                 range: +9.99E+62 < N < +9.99E-65
                                 accuracy: 14 decimal digits

Refer to Chapter 3 for a more complete discussion
of internal machine representation.

15.5   PRINT and INPUT

PRINT and INPUT write and read in ASCII format.
These instructions are primarily intended to write
data to an output file which is to be printed or
typed at a later time, and to read a data file
which was created by the editor or another system.
When used with a file, they exactly parallel their
operation with the console terminal. In
particular, file INPUT requires a comma or carriage
RETURN between INPUT data items.

IT IS RECOMMENDED THAT THE PUT AND GET INSTRUCTIONS
BE USED WITH DATA FILES WHICH ARE TO BE ACCESSED BY
BASIC PROGRAMS.

The PRINT and INPUT instructions translate numeric
data from ASCII to Internal Machine Format (or vice
versa). They are therefore slower than the PUT and
GET instructions. There is little need for the
programmer to keep track of the type of variable
which is written out with a PRINT instruction. An
integer variable which is written using the PRINT

instruction may be INPUT back into a string or
floating point variable without losing its meaning.
No attempt should be made to INPUT an ASCII string
into a numeric variable. A file which has been
output using the PRINT instruction may also be
TYPEd out and read whereas the same file output
with the PUT instruction would not be able to be
TYPEd.

15.6 PUT and GET

The PUT and GET instructions write and read in
internal machine format. If an integer variable is
written out to a file using a PUT instruction, it
must be read back into an integer variable (and not
into a floating point variable or an ASCII string)
using a GET instruction or it will be meaningless.
The machine does no translation with these
instructions so they can be executed faster than
PRINT and INPUT. The programmer must keep track of
the types and lengths of variables written by PUT
so that they may be read back in properly.

15.7 An Interesting Note

A string (which is stored internally in ASCII code)
may be output with a PUT or PRINT instruction with
the same general result (except that PRINT outputs
a carriage RETURN-line feed sequence where
appropriate and tabs, using spaces, when variables
are separated by commas). This is because PUT will
output a string in internal machine format, which
(for a string) is ASCII. The PRINT will not
translate the string because it is already in ASCII
format.

15.8   File Pointer

While a file is OPEN, BASIC maintains a pointer
which may determine where the next read or write
will occur. This pointer can be manipulated by the
various file read and write instructions as will be
explained in the following paragraphs.

When a file is first OPENed, the File Pointer is
positioned just before the first byte of the first
record of the file. If the file is a new file
(i.e., it contains no data), the beginning of the
file and the end of the file coincide, and so the
File Pointer also points to the end of file. This
is appropriate. If a read is attempted on a new
file, an end of file error will be returned because
there is no data in the file.

If no record number is specified in a file I/O
instruction (sequential I/O), the data list
contained in the instruction is written to or read
from the file from the current position of the file
pointer.

If a record number is specified in a file I/O
instruction (random I/O), the File Pointer is moved
to a position just before byte zero of the
specified record and then the data list contained
in the instruction is written to or read from the
file from the new position of the File Pointer.

If a byte number is specified in addition to the
record number, the File Pointer will be positioned
just before the specified byte in the current or
specified record before the I/O instruction is
executed.

If there is no data list associated with a PUT
instruction, the pointer is repositioned as
specified above and no input or output takes place.

After the input or output has taken place, the PUT
and GET instructions leave the File Pointer just
after the byte which was last input or output.

The PRINT and INPUT instruction use the Carriage
RETURN or Carriage RETURN-Line Feed sequence as a
delimiter.

The PRINT instruction will move the File Pointer to
a position just after the last PRINT character. If

164

the PRINT line starts with byte zero of the current record and is the same length as the specified record size, the pointer will be positioned just before byte zero of the next logical (sequential) record.

The PRINT instruction will automatically insert a Carriage RETURN/Line Feed sequence if:

1. the data list associated with the PRINT instruction will cause a record to be output which is longer than the page width (refer to the SET instruction), and

2. no one item in the data list is not itself longer than the specified page width.

This can happen when items in a PRINT list are separated by commas or a PRINT instruction is terminated with a comma or semicolon and a subsequent PRINT instruction adds to the already PRINTed line. This rule also applies to the PRINT instruction when used to send output to the console.

A PRINT list containing an item which is longer than the page width will cause a run time error. The default page width is 80 characters. It is possible to change this parameter by using the SET instruction.

15.9   Sequential Files

A sequential file is written in the order of the record numbers. That is, record number zero is written with the first output (PUT or PRINT) instruction, record number one is written next, then record number two, etc. This continues until the file is CLOSEd. When the file is OPENed again, the first record which is output will write over record number zero, etc.

When a file is read sequentially, record zero is read first, then record one, etc.

Note that if an entire record is not input or output each time the file is accessed, the File Pointer will remain after the last character which was read or written. Any subsequent file access

165

will read or write from that position, not from the
beginning of the next record.

15.10   Random Files

A random access file is written in the order
specified by the programmer. Each output
instruction must specify the number of the record
which is to be written. When a file is read
randomly, the programmer must specify the record
number to be read for each INPUT or GET
instruction.

A file which was written to sequentially may be
read as a random file. Files which were written to
randomly may be read by sequential or random
instructions provided that no attempt is made to
read a record before it has been written.

A file may be accessed by any combination of
sequential and random instructions.

A random instruction, one that specifies a record
or record and byte number, will reposition the File
Pointer before accessing the file.

A sequential instruction, one that does not specify
a record number, will access the disk from the
current position of the File Pointer.

15.11   CDOS DUMP Utility

The CDOS DUMP utility program is very useful for
determining exactly what is being sent to a file.
The user must exit from BASIC in order to use this
program. Refer to the CDOS manual, Utility
Programs - DUMP, for further information.

instruction: CREATE

format: [Ln] CREATE file-ref

where:

Ln   is an optional line number

file-ref includes the 1 to 8 character
          file name and optionally a disk
          drive specifier and file name
          extension.

The CREATE instruction places the file name (and
file name extension) in the directory of the
specified disk. A file may only be created once,
and must be CREATEd before it can be OPENed.

Notes:

1.   The disk drive specifier and/or the file name
     and/or the file name extension may be a string
     variable or a string literal.

2.   The file name and file name extension may
     include any printable ASCII character except
     the following:

         $ * ? = / . , : - "space"

3.   No space is allocated to a file by the CREATE
     instruction. All file space under CDOS is
     dynamically allocated only when and where
     needed.

4.   Error Number 137 (File Already Exists) will
     result if the file already exists in the
     directory when the CREATE instruction is
     given.

instruction: OPEN

    format: [Ln] OPEN\n\ file-ref

           [Ln] OPEN\n,pl\ file-ref

           [Ln] OPEN\n,pl,p2\ file-ref

where:

Ln   is an optional line number

n   is the required file (channel) number.

pl   (for a disk file) is the optional record size in bytes which may be assigned any value between 1 and 32,767. The default value is 128 bytes per record (one sector).

p2   (for a disk file) is the optional file access mode specifier:

               =1 is read only
               =2 is write only
               =3 is read and write
                   (default value)

file-ref includes the 1 to 8 character file name and optionally a disk drive specifier and file name extension.

The OPEN instruction allows a disk file or system device to be linked to a file number (channel) for future reference in connection with file input/output instructions (i.e., PUT, GET, INPUT, PRINT, CLOSE).

Notes:

1.   The disk drive specifier and/or the file name and/or the file name extension may be a string variable or a string literal.

2.   The file name and file name extension may include any printable ASCII character except the following:

$ * ? = / . , : - "space"

3.   The file number must be between 1 and the
        maximum channel number available. As disk
        BASIC is shipped, the maximum channel number
        is 4. See the Appendix for the method of
        changing the number of channels available.

4.   Error number 133 (File Number) will result if
        an attempt is made to reference a file number
        greater than the maximum channel number
        available.

5.   The file number Ø (zero) is reserved for the
        console. It cannot be OPENed by the user.
        All Input/Output (GET, INPUT, PUT, and PRINT)
        directed from/to file Ø will use the console.

        Caution, if file number Ø is specified by the
        CLOSE instruction, all currently OPENed files
        will be CLOSEd.

6.   Although files OPENed for read/write access
        may be used as write only files, slightly
        faster execution speed may result if the file
        is OPENed for write only access.

7.   If the PRINT instruction is used to write a
        single item which is longer than the current
        page width, it will be necessary to increase
        the page width (using the SET instruction) in
        order to avoid Error number 6 (PRINT ITEM
        SIZE). Refer to the discussion section of
        this chapter for additional information.

instruction: CLOSE

    format: [Ln] CLOSE

             [Ln] CLOSE \n\

where:

Ln   is an optional line number

n   is an optional file (channel) number. The default value is ALL currently OPENed files.

The CLOSE instruction disassociates the channel number and file which were associated by the OPEN instruction.

Note:

If the CLOSE instruction is given with the file number set equal to Ø (zero), all currently OPENed files will be CLOSED.

instruction: PRINT

format: [Ln] PRINT

[Ln] PRINT exp-1,...,exp-n

[Ln] PRINT\n\

[Ln] PRINT\n,p1\

[Ln] PRINT\n,p1,p2\

[Ln] PRINT\n\ exp-1,...,exp-n

[Ln] PRINT\n,p1\ exp,...,exp-n

[Ln] PRINT\n,p1,p2\ exp-1,...,exp-n

where:

Ln   is an optional line number

n   is an optional file number. If
          no file number is specified, or
          if the file number is zero,
          output goes to the console
          terminal.

p1   (for a disk file) is the
          optional record number. The
          default value is sequential
          access starting with record 0.

p2   (for a disk file) is the
          optional byte number. The
          default value is byte 0.

exp1-n   is an optional list of numeric
          or string expressions, numeric
          or string variables, or string
          literals. The data list must be
          separated by either semicolons
          or commas. If an item is
          followed by a comma, the item
          following it will be printed
          starting in the next tab
          position; if it is followed by a
          semicolon no space will be left
          before printing the next item.

When used without a file reference, the PRINT
instruction will cause the data list to be output

171

to the console. When used with a file reference,
it will cause the data list to be output to an
ASCII device (e.g., the line printer) or a disk
file in ASCII format.

Notes:

IT IS RECOMMENDED THAT PRINT ONLY BE USED FOR THE
OUTPUT OF PRINT OR LIST FILES AND NOT BE USED FOR
WRITING DATA FILES WHICH ARE TO BE READ BACK BY A
BASIC PROGRAM.

1.   The at (@) sign may be used to abbreviate the
      word PRINT in this instruction.

2.   Most of the above forms of the PRINT
      instruction may incorporate the PRINT USING
      feature. Refer to PRINT USING for additional
      information.

3.   The PRINT instruction outputs a carriage
      return-line feed (ØD, ØA hex) sequence at the
      end of the data list. If the file is to be
      read in using the INPUT instruction, a CR-LF
      sequence (or just a CR) must follow each item
      which is output, as it is this character or
      character sequence which delimits (terminates)
      each item. In BASIC, there are two ways this
      can be accomplished:

      a.   Allow only one item per PRINT
            instruction.

      b.   Insert a CR between each item as follows:

            100 A$ = CHR$(13)
            200 PRINT \1\ "STRING"; A$; N1; A$; Q$

      Statement 100 assigns the hexadecimal value of
      the CR to the string variable A$. Statement
      200 PRINTs a string literal, a numeric
      variable, and a string variable, each
      separated by the CR.

      The above procedure will allow all of the
      items output by the PRINT instruction to be
      read by the INPUT instruction which looks for
      a CR between each item read.

      If only numeric data is to be PRINTed and

INPUT a comma (ASCII 44) may be used as a delimiter between variables within one INPUT list. The last item in the data list must be delimited by a CR.

4. Specifying a negative number for either p1 or p2 will result in the default value being assigned to that parameter.

5. The PRINT instruction, when given without a data list, causes one blank line to be sent to the console or a PRINT file.

instruction: INPUT

format: [Ln] INPUT var-1,...,var-n

[Ln] INPUT\n\ var-1,...,var-n

[Ln] INPUT\n,pl\ var,...,var-n

[Ln] INPUT\n,pl,p2\ var-1,...,var-n

where:

Ln   is an optional line number

n   is an optional file number. If
            no file number is specified, or
            if the file number is zero,
            input is received from the
            console terminal.

pl   (for a disk file) is the
            optional record number. The
            default value is sequential
            access starting with record 0.

p2   (for a disk file) is the
            optional byte number. The
            default value is byte 0.

varl-n   is a list of one or more numeric
            or string variables.

When used without a file reference, the INPUT
instruction will cause the data list to be INPUT
from the console. When used with a file reference,
it will cause the list to be INPUT from an ASCII
device (e.g., paper tape reader) or a disk file in
ASCII format.

Notes:

1.   When a string variable is used with an INPUT
     instruction, the portion of the string which
     is referenced is set equal to null characters
     before the source is moved into the string
     variable. The whole string is referenced if
     no subscripts follow the string variable,
     while various substrings may be referenced by
     the use of subscripts. Refer to section
     5.2.3, Referencing String Variables, for a
     complete discussion of the subject. The INPUT

instruction will not move more characters than
can be accepted by the destination string or
substring which is being referenced.

2. INPUT can accept no more than 132 characters
    per line.

3. INPUT retrieves only 7 bit ASCII. The GET
    instruction must be used to retrieve all 8
    bits.

4. INPUT treats some control characters as
    editing or end of file commands.

instruction: PUT

format: [Ln] PUT\n\

[Ln] PUT\n\ exp-1,...,exp-n

[Ln] PUT\n,p1\ exp,...,exp-n

[Ln] PUT\n,p1,p2\ exp-1,...,exp-n

where:

Ln   is an optional line number

n   is a file number. If file number 0 is used, output will be set to the console.

p1   (for a disk file) is the optional record number. The default value is sequential access starting with record 0.

p2   (for a disk file) is the optional byte number. The default value is byte 0.

expl-n   is an optional list of numeric or string expressions, numeric or string variables, or string literals.

The PUT instruction outPUTs the data in the data list (exp-1,...exp-n) to the file specified by the file number (n). The data is output in internal machine format. This is useful if the data is to be read back by BASIC. The PRINT instruction should be used if the file is to be TYPEd or listed to the printer.

Notes:

1.   If no outPUT list is included, only the device status is set (i.e., record and byte position on a disk file). This is a useful way of setting status (position) without actually initiating any data transfer.

2.   The PUT instruction will output data to a file in internal machine format (see notes at the beginning of this chapter on Internal Machine

vs. ASCII Representation). Numeric data which
has been output using the PUT instruction
<u>must</u> be read back in using the GET
instruction.

3.   ASCII data which is output by the PUT
instruction will not be readable by INPUT
unless a CR appears at least every 132 bytes.

instruction: GET

format: [Ln] GET\n\ exp-1,...,exp-n

[Ln] GET\n,pl\ exp,...,exp-n

[Ln] GET\n,pl,p2\ exp-1,...,exp-n

where:

Ln   is an optional line number

n   is a file number. If file
          number Ø is used, input will be
          accepted from the console.

pl   (for a disk file) is the
          optional record number. The
          default value is sequential
          access starting with record Ø.

p2   (for a disk file) is the
          optional byte number. The
          default value is byte Ø.

expl-n   is an optional list of numeric
          or string expressions, numeric
          or string variables, or string
          literals.

The GET instruction GETs the data in the data list
(exp-1,...exp-n) from the file specified by the
file number (n).   The data is input without
translation (internal machine format).   This is
useful if the data has been outPUT by the PUT
instruction.

Note:

1.   When a string variable is used with a GET
      instruction, the portion of the string which
      is referenced is set equal to null characters
      before the source is moved into the string
      variable.   The whole string is referenced if
      no subscripts follow the string variable,
      while various substrings may be referenced by
      the use of subscripts.   Refer to section
      5.2.3, Referencing String Variables, for a
      complete discussion of the subject.   The GET
      instruction will not move more characters than
      can be accepted by the destination string or
      substring which is being referenced.

PROGRAMMED FUNCTIONS

Cromemco 16K Extended BASIC includes a number of arithmetic and trigonometric functions which perform common, frequently used calculations. These functions are pre-defined in BASIC so that the programmer does not need to write a program everytime one of these functions is required.

Cromemco BASIC also includes a number of functions designed to increase string handling capabilities, a number of system functions which provide general system information, and a function which makes it possible to call assembly language subroutines.

In addition to these pre-defined functions, Cromemco BASIC permits the programmer to define up to 26 additional functions.

Cromemco 16K BASIC includes the following functions:

16.1   Arithmetic Functions

ABS (X)	absolute value of X
BINAND(X,Y)	binary logical AND
BINOR(X,Y)	binary logical OR
BINXOR(X,Y)	binary logical EXCLUSIVE OR
EXP (X)	the value "e" to the power X
FRA (X)	the fractional portion of X
INT (X)	integer value of X
IRN (X)	generates an integer random number between 0 and 32767
LOG (X)	natural logarithm of X

179

MAX (X1,.,Xn) returns the numeric expression Xn with the maximum value in the expression list

MIN (X1,.,Xn) returns the numeric expression Xn with the minimum value in the expression list

RANDOMIZE used with RND and IRN to produce different sets of random numbers

RND (X) generates a random number between Ø and 1

SGN (X) algebraic sign of X

SQR (X) square root of X

16.2 Trigonometric Functions

ATN (X) arctangent of X

COS (X) cosine of X

SIN (X) sine of X

TAN (X) tangent of X

16.3 Programmer Defined Functions

DEF FNs (X1,...,Xn) = Y
 allows the user to define up to 26 different functions

16.4 String Functions

ASC (X$) provides equivalent ASCII numeric value of the first character of X$

CHR$ (X) gives a single character string which is the ASCII equivalent of X

LEN (X$)	returns the length of string X$.
POS (X$,Y$,n)	determines the location of a substring (Y$) within a string (X$) by returning the value of the position at which the first character of the substring is located starting with character n.
STR$ (X)	converts any numeric expression X to a string which is the character representation of X.
VAL (X$)	converts any string expression X$ to the numeric representation of X$.

Descriptions and examples of each of these functions are included in the following section.

Also see Chapters 17 and 18 for descriptions of additional functions.

function: ABSolute value

format: ABS (aexp)

where:

aexp   is an arithmetic expression,
variable, or constant.

The ABS function gives the absolute (i.e.,
positive) value of X, which can be any numeric
expression.

Example:

>>LIST

    10 PRINT ABS(-26), ABS(26)
    20 END

>>RUN
26   26
20 END

functions: BINAND

BINOR

BINXOR

format: BINAND(aexp-1,aexp-2)

BINOR(aexp-1,aexp-2)

BINXOR(aexp-1,aexp-2)

where:

aexp1-2 are arithmetic expressions, variables, or constants.

The BINAND, BINOR, BINXOR functions perform logical operations bit by bit on 16-bit operands.

BINAND performs a BINary AND logical operation on aexp-1 and aexp-2. It returns a 1 bit in a given position if both bits are equal to 1 and a 0 otherwise.

BINOR performs a BINary OR logical operation on aexp-1 and aexp-2. It returns a 1 bit in a given position if either bit is equal to 1 and a 0 otherwise.

BINXOR performs a BINary exclusive OR logical operation on aexp-1 and aexp-2. It returns a 1 bit in a given position if either bit is equal to 1 and the other bit is equal to 0. A 0 is returned otherwise.

Notes:

1.   If necessary, aexp-1 and/or aexp-2 will be converted to 16 bit integers for the purpose of these functions.

2.   Refer to section 6.4 for a discussion of boolean operators.

function: EXPonent

format: EXP (aexp)

where:

aexp   is an arithmetic expression,
            variable or constant.

The EXP function calculates the value of the
constant e (where e = 2.71828...) raised to the
aexpth power.

Example:

>>LIST

10 A=4.1
20 B=EXP(A)
30 PRINT B
40 END

>>RUN
60.340287597344
40 END

function: FRActional portion

format: FRA (aexp)

where:

aexp   is an arithmetic expression,
variable or constant.

The FRA function returns the fractional portion of
aexp.

Example:

>>LIST

10 B=3.7
20 PRINT FRA(B)
30 END

>>RUN
0.7
30 END

185

function: INTeger portion

format: INT (aexp)

where:
           aexp   is an arithmetic expression,
                     variable or constant.

The INT function returns the largest integer value
which is less than or equal to aexp.

Examples:

```
>>LIST

  100 A=5.7
  200 PRINT INT(A)
  300 END

>>RUN
5
***300 END***
>>PRINT INT (-5.7)
-6
```

function: Integer Random Number generator

format: IRN (X)

where:
X   is a dummy argument.

The IRN function generates an integer random number
between 0 and +32767.

Note:

1.   To change the sequence of random numbers, the
     RANDOMIZE instruction should be included in
     the program.

Example:

>>LIST

    10   FOR N=1 TO 10
    20   PRINT IRN(6)
    30   NEXT N
    40 END

>>RUN
29284
25801
18835
4647
9295
18846
4924
10105
20210
7652
40 END

The program listed above will print out 10 integer
random numbers. If this program is run a second
time, it will generate the same 10 random numbers.
A different set of random numbers will be generated
only if a RANDOMIZE statement is included at the
beginning of the program.

function: LOGarithm

format: LOG (aexp)

where:

aexp   is an arithmetic expression,
            variable or constant.

The LOG function calculates the natural logarithm
(i.e., log to the base e) of aexp.

Note:

1.   The logarithm base 10 for aexp can be computed
      as follows:

      log base 10(aexp) = LOG (aexp) / LOG (10)

Example:

>>LIST

    10 INPUT A
    20 B=LOG(A)
    30 PRINT B
    40 END

>>RUN
? 3.2
1.1631508098056
40 END

function: MAXimum value

format: MAX (aexp-1,...,aexp-n)

where:

aexpl-n   are   arithmetic   expressions,
variables, or constants.

The MAX function examines the list of arithmetic
expressions (aexp-1 through aexp-n) and returns the
value of the largest expression.

Example:

>>LIST

10 X=10
20 Y=25
30 M=MAX (X,Y)
40 PRINT M
50 END

>>RUN
25
50 END

function: MINimum value

format: MIN (aexp-1,...,aexp-n)

where:

aexp1-n are arithmetic expressions,
variables, or constants.

The MIN function examines the list of arithmetic
expressions (aexp-1 through aexp-n) and returns the
value of the smallest expression.

Example:

>>LIST

10 X=5
20 Y=10
30 M=MIN(X,Y)
40 PRINT M
50 END

>>RUN
5
50 END

instruction: RANDOMIZE

format: [Ln] RANDOMIZE

where:

Ln   is an optional line number.

The RANDOMIZE instruction is used to reset the
random number dummy variable used by the RND and
IRN functions so that a different sequence of
random numbers will be produced each time the RND
and IRN functions are used.

Notes:

1.   RANDOMIZE should be used only once within a
     program utilizing RND to ensure that a truly
     random sequence of numbers results.

2.   RANDOMIZE shold be used after generating every
     1000 numbers using IRN. This will ensure a
     truly random sequence of numbers.

Example:

>>LIST

10 PRINT "THIS IS A RANDOM NUMBER ";
20 RANDOMIZE
30 PRINT RND (0)
40 END

>>RUN
THIS IS A RANDOM NUMBER 0.7137712225
40 END
>>RUN
THIS IS A RANDOM NUMBER 0.8171978025
40 END

This program will print a different random number
everytime the program is RUN.

function: RaNDom number generator

format: RND (X)

where:

X   is a dummy argument.

The RND function generates a random number in the range $0 \leq RND(0) < 1$.

Examples:

    >>LIST

        10 FOR N=1 TO 100
        20 PRINT RND(2)
        30 NEXT N
        40 END

The above example will print out 100 random numbers. If this program is run a second time, it will generate the same 100 random numbers. To generate a new set of random numbers each time the program is run, the above example can be rewritten as follows:

    >>LIST

        10 RANDOMIZE
        20 FOR N=1 TO 100
        30 PRINT RND(2)
        40 NEXT N
        50 END

function: SiGN

format: SGN (aexp)

where:

aexp   is an arithmetic expression,
           variable or constant.

The SGN function returns a +1 if the value of the
expression aexp is greater than Ø, a Ø if aexp
equals Ø, and a -1 if aexp is less than Ø.

Example:

        >>LIST

            1Ø INPUT A,B,C
            2Ø PRINT SGN(A)
            3Ø PRINT SGN(B)
            4Ø PRINT SGN(C)
            5Ø END

        >>RUN
        ? -12,Ø,14
        -1
        Ø   .
        1
        5Ø END

function: SQuaRe root

format: SQR (aexp)

where:

aexp   is an arithmetic expression,
          variable or constant.

The SQR function calculates the square root of any
positive expression.

Example:

```
>>PRINT SQR(9)
3
```

function: ArcTaNgent

format: ATN (aexp)

where:

aexp   is an arithmetic expression,
variable or constant.

The ATN function calculates the arctangent of
aexp.

Note:

1.   Although 16K BASIC does not include predefined
ARCSIN and ARCCOS functions, the user can
calculate these using the ATN function as
follows:

ARCSIN (X) = ATN (X/SQR(-X*X+1))
ARCCOS (X) = -ATN (X/SQR(-X*X+1))+2. *ATN(1.)

Example:

>>PRINT ATN(.80)
0.67474094222353

In this example the arctangent of .80 is equal to
.6747094222353 radians.   If the DEGree mode is
selected, the arctangent of .80 is given in
degrees:

>>DEG

>>PRINT ATN(.80)
38.659808254087

function: COSine

format: COS (aexp)

where:

aexp   is an arithmetic expression,
           variable or constant.

The COS function calculates the cosine of the angle
represented by aexp.

Notes:

1.   See ATN for a description of how to calculate
     ARCCOS.

2.   Unless DEG mode has been selected, it is
     assumed that the value of aexp is expressed in
     RADians.

Example:

    >>LIST

        10 INPUT A
        20 LET B=A*2
        30 PRINT COS (B)
        40 END

    >>RUN
    ? .60
    0.36235775447529
    40 END

In this example, the cosine of a 1.20 radian angle
is .36235775447529.

function: SINe

format: SIN (aexp)

where:

aexp   is an arithmetic expression,
        variable or constant.

The SIN function calculates the sine of an angle
represented by aexp.

Notes:

1. See ATN for a description of how to calculate
    ARCSIN.

2. It is assumed that the value of aexp is
    expressed in RADians unless the DEG mode has
    been specified.

Examples:

```
>>LIST

   10 INPUT A
   20 PRINT SIN(A*3)
   30 END

>>RUN
? .04
0.11971220728892
***30 END***
```

In this example, the sine of a .12 radian angle is
approximately .12.

In the following example, we first select the DEG
mode and then request the program to print the sine
of a 90 degree angle:

```
>>LIST

    5 DEG
   10 PRINT SIN(90)
   20 END

>>RUN
1
***20 END***
```

The sine of a 90 degree angle is 1.0.

197

function: TANgent

format: TAN (aexp)

where:

aexp   is an arithmetic expression,
            variable or constant.

The TAN function calculates the tangent of the
angle represented by aexp.

Note:

1.   It is assumed that the value of aexp is
     expressed in RADians unless the DEGree mode
     has been specified.

Example:

```
>>LIST

    5 DEG
   10 PRINT TAN(30)
   20 END

>>RUN
0.577350269189
***20 END***
```

function: Programmer Defined

format: DEF FNs(avar-1,avar-2,...,avar-n)=aexp

where:

s   is any single ASCII character
            (A-Z).

avar1-n   are arithmetic variables.

aexp   is an arithmetic expression,
            variable, or constant.

The DEF function permits the programmer to define
up to 26 functions in addition to the pre-defined
functions included in BASIC.

Notes:

1.   The definition of a function must be a
     statement which is encountered during the
     execution of a program in order for BASIC to
     retain the definition. After the function has
     been defined, it may be used in a command
     line.

2.   Any of the variables used in the definition of
     a function (avar-1 through avar-n) are local
     to the function. Their use in the definition
     does not conflict with a variable of the same
     name appearing elsewhere in the program.

     If a variable appears on the right side of the
     definition without appearing on the left, it
     is a global variable and maintains its value
     through a function call.

Examples:

```
>>LIST

    10  X=20
    20  Y=10
    30  Z=5
    40  DEF FNR(A,B,C)=A*B+C
    50  PRINT FNR (X,Y,Z)
    60  END

>>RUN
205
***60 END***
```

In the above example, variables X, Y, and Z are global variables while A, B, and C are local to the definition of the function.

The following example should help to clear up any confusion between local and global variables.

```
>>LIST

    100  DEF FNZ (P) = P * D
    200  P = 77
    300  D = 5
    400  E = 10
    500  PRINT FNZ (E)
    600  PRINT P
    700  END

>>RUN
50
77
***700 END***
```

In the above example, statement 100 defines the function FNZ in terms of the formal parameter (function definition variable) P and the program variable D. In statement 100 P is called a function definition variable because it appears to the left of the equal sign in the function definition. As such, P can not be accessed by the user. The program variable (P) defined by statement 200 is a different variable than the aforementioned function definition variable, and can be accessed by the user. According to the above definitions, D, appearing in statement 100, is a program variable. In statements 300, 400, and 500 program variables P, D, and E are assigned

values.

In statement 5ØØ, the value of the function FNZ is
computed using the value of E to replace P in the
definition of the function. Being a program
variable, D maintains its value in the calculation
of the value of the function. Once the value of
the function has been calculated, its value is
printed.

Statement 6ØØ prints the value of the program
variable P to demonstrate that the program variable
P can be accessed by the user, that it maintains
its value through the function call, and that it is
a different variable than the function definition
variable P.

function: ASCii value of a character

format: ASC (svar)

where:

svar   is a string variable or literal.

The ASC function gives the ASCII decimal value of
the first character of any string expression.

Note:

1.   See the Appendix for a table of ASCII
      characters and their values.

Example:

```
>>LIST

   10  INPUT X$
   20  PRINT ASC(X$)
   30  END

>>RUN
? A
65
***30 END***
```

In this example, the ASCII decimal value for the
first character of the string variable X$
(character A) is 65.

function: CHaRacter

format: CHR$ (aexpr)

where:

aexpr   is an arithmetic expression,
           variable, or constant.

The CHR$ function returns the single character
which is represented in ASCII by aexpr.

Notes:

1.   If aexpr is outside of the range $0 \leq aexpr \leq 255$
     an error will be generated.

2.   This function allows the user to draw graphs
     or figures with special characters. The
     function may also be used to initiate special
     functions such as cursor positioning,
     generating line or form feeds, or causing a
     bell to sound on the terminal.

3.   This function may be used to output non-
     printing and special purpose characters such
     as control or underline characters.

Example:

        >>LIST

           10 INPUT X
           20 PRINT CHR$(X)
           30 END

        >>RUN
        ? 42
        *
        30 END

The ASCII decimal value 42 is equivalent to the
character *. Thus, in the above example, the
instruction PRINT CHR$ (42) instructs the computer
to output the character * on the terminal.

function: LENgth of string

format: LEN (svar)

where:
         svar   is a string variable or literal.

The LEN function returns an integer value which is
equal to the number of characters in any string
variable X$.   In other words, the LEN function
gives the length of a string.

Note:

1.   Both characters and spaces are counted as part
     of the length.   Trailing null characters are
     not counted as part of the string length.

Example:

```
>>LIST

    10  DIM X$(20),Y$(30)
    20  INPUT X$,Y$
    30  @"THERE ARE "; LEN(X$); " CHARACTERS IN X$"
    40  @"THERE ARE "; LEN(Y$); " CHARACTERS IN Y$"
    50  END

>>RUN
? EXAMPLE
?? LENGTH COMMAND
THERE ARE 7 CHARACTERS IN X$
THERE ARE 14 CHARACTERS IN Y$
***50 END***
```

function: POSition

format: POS (svar-1,svar-2,,aexp)

where:

svar-1   is a string variable or literal

svar-2   is a substring variable or literal

aexp   is an arithmetic expression, variable, or constant

The POS function is used to locate the position within a string (X\$) of the substring (Y\$). The position within the string X\$ at which the search is to begin is specified by the arithmetic expression aexp. This function gives a value equal to the position of the first character of the substring within the string.

Notes:

1.   A -1 is returned if the substring is not found.

2.   Remember that the first element of a string is numbered 0.

Example:

>>LIST

```
10 DIM X$ (50)
20 X$="THIS IS A SUBSTRING SEARCH"
30 P=POS(X$,"IS",4)
40 R=POS(X$,"R",20)
50 PRINT P
60 PRINT R
70 END
```

>>RUN
5
23
70 END

In this example, the computer is first instructed to search for "IS" starting from the fourth

character in X$ and second to search for "R"
starting from the twentieth character in X$.
Starting from position 4, the first character in
substring "IS" is located in position 5. Starting
from position 20, the first character "R" is
located in position 23. Consequently, the computer
returns a value of 5 for P and 23 for R.

A very useful application of this function is the
conversion of numbers from hexadecimal to decimal.
The following program demonstrates the principle
using a one digit number:

```
>>LIST

1000 DIM A$(0)
1100 INPUT "One digit hex number:  ",A$
1200 D = POS ("0123456789ABCDEF", A$, 0)
1300 IF D = -1 THEN GOTO 1100
1400 PRINT "The decimal equivalent is:  ";D
1500 GOTO 1100
1600 END
```

function: STRing

format: STR$ (aexp)

where:

aexp   is an arithmetic expression,
          variable, or constant.

The STR$ function converts aexp to a string which
is the ASCII representation of the expression.

Note:

1.   Valid input to this function include the
     decimal point (.) and a leading plus (+) or
     minus (-) sign.

Example:

>>LIST

    10 INPUT A
    20 A$=STR$(A)
    30 PRINT A$
    40 END

>>RUN
? 8.45
8.45
40 END

function: VALue

format: VAL (svar)

where:

svar   is a string variable or literal.

The VAL function converts a string (svar) into a numeric variable which may be used in arithmetic expressions.

Notes:

1.   If the argument string for VAL consists of both numeric and non-numeric information, the following conventions hold:

a.   If the first character is non-numeric, VAL will return a zero value (this can be an extremely useful way of decoding a user's input). The first character is considered to be numeric if it is the leading percent sign (%) of a legal hexadecimal constant.

b.   If the first character or characters are numeric they will be converted without consideration of the portion of the string after the first non-numeric character.

Example:

>>LIST

```
10 X$="26.6321"
20 A=1
30 B=A+VAL(X$)
40 PRINT B
50 END
```

>>RUN
27.6321
50 END

<div style="text-align:center">

command: DiSK

format: DSK
           DSK "X"

</div>

where:

      X   is a disk drive specifier (1, 2, 3, 4, A, B, C, or D) or the at sign or zero (@ or 0).

The DSK instruction displays or alters the current system default disk drive.

Notes:

1.   When used without a disk drive specifier, DSK displays the current disk drive identifier.

       While a program is RUNning, DSK will not display anything. Use IOSTAT (0,0) to determine the current drive while in the RUN mode.

2.   When used with a disk drive specifier, DSK changes the current system default disk drive. The following is a list of equivalent disk drive specifiers:

          A: or 1
          B: or 2
          C: or 3
          D: or 4

       The above specifiers may be enclosed in quotation marks (string literal) or assigned to a string variable.

       New disks are not logged in by changing the default disk drive. See the following note for the method of logging in a new disk from BASIC.

3.   When used with the at sign or zero (@ or 0), DSK logs in a new disk (after it has been inserted in any drive) and causes drive A to become the current system default disk drive. The effect of this is similar to typing CTRL-C while in CDOS.

instruction: ECHO

format: [Ln] ECHO

where:

Ln   is an optional line number.

The ECHO instruction is used to re-enable the display of certain information at the console terminal after the display has been disabled by the NOECHO instruction.

instruction: NO ECHO

format: [Ln] NOECHO

where:

Ln   is an optional line number.

The NOECHO instruction is used to disable the display of user entered responses to the INPUT instruction.

Notes:

1.   This instruction is useful when a secret code or password is to be INPUT. The code will not be displayed on the screen, making theft of the code almost impossible.

2.   The NO ECHO mode is NOT reset by the RUN or SCRatch instructions.

Example:

```
>>LIST

   100 NOECHO
   200 INPUT X
   300 PRINT X
   400 END

>>RUN
?          (user  entered  INPUT  is  not
           displayed)
517        (the value of X is PRINTed by the
           PRINT statement)
***400 END***
```

instruction: ERASE

format: [Ln] ERASE svar

where:

Ln   is an optional line number.

svar   is a string literal or string
            variable file reference.

The ERASE instruction will remove a file from the
file directory.

Examples:

        ERASE "TEMP.BAS"
        ERASE "*.BAK"
        ERASE "B:TEMP.REL"

The first of these examples will ERASE the file
named TEMP.BAS from the current or default disk
drive. The second will ERASE all files from the
current disk drive with the file name extension of
BAK. The third example will erase the file called
TEMP.REL from the disk in the B drive.

instruction: ESCape

format: [Ln] ESC

where:

Ln   is an optional line number.

The ESC instruction is used to re-enable ESCape key
operation after it has been disabled by the NOESC
instruction.

instruction: NOESCape

format: [Ln] NOESC

where:
        Ln   is an optional line number.

The NOESCape instruction disables the console
terminal escape key operation.

Notes:

1.   Most terminal keyboards include a key labeled
      ESC. BASIC recognizes the ESCape character as
      a signal to abort program execution and return
      to the command mode. The NOESC instruction is
      used to prevent program interruption when the
      ESC key is pressed.

2.   ESCape results from the use of the CTRL-Z,
      ESC, or CTRL-[key.

function: FREe space

format: FRE (X)

where:

X   is a dummy argument,

The FRE function gives the number of bytes of memory in the User Area which are currently free or unused.

Notes:

1.   Certain statements do not occupy their full space until after they have been executed (e.g., DIM).

2.   Because BASIC allocates space for its internal tables in segments, the FRE function is only an approximation of the actual number of bytes available to the user.

3.   Space recovered from DELETEd lines is available for new program lines, not for arrays or variables. Thus, the FRE function will not necessarily reflect available program space.

      If statement lines have been DELETEd from the User Area, it will be necessary to perform the following procedure in order for the FRE function to reflect the change:

      a)   LIST the program (do not SAVE it) to a temporary disk file.

      b)   SCRatch the User Area.

      c)   ENTER the temporary disk file.

      d)   Type PRINT FRE(X).

function: IOSTATus

format: [Ln] IOSTAT (aexpr-1, aexpr-2)

where:

Ln   is an optional line number.

aexpr-1   is a file (channel) number.

aexpr-2   is the status parameter.

The IOSTAT function returns the current status of an opened BASIC file.

Status Parameter	Returned Value	Meaning
0	0	File is ok.
	1	End of File.
	2	Attempt to read an unwritten segment during a random access read.
1	X	X is the current sector number. (This is not the record number.)
2	Y	Y is the current byte number within sector X (above).

Notes:

1.   Devices other than disk drives may or may not return status values.

2.   IOSTAT(0,0) will return the current disk number. Disk A=1, B=2, C=3, and D=4.

instruction: ON ERROR

       format: [Ln] ON ERROR STOP

                [Ln] ON ERROR GOTO nl

                [Ln] ON ERROR GOSUB nl

where:

    Ln   is an optional line number.

    nl   is the line number of the
                statement to which control is
                transferred.

The ON ERROR instruction causes program control to
be transferred as specified (STOP, GOTO, or GOSUB)
when a non-fatal error occurs during program
execution.

Notes:

1.   A non-fatal error in BASIC is any error listed
     in the error table (see Appendix A) with a
     number of 128 or greater. Errors numbered 127
     and below are defined as fatal errors and
     cannot be trapped with an ON ERROR statement.

2.   If ON ERROR is written at the beginning of a
     program, the instruction specified with ON
     ERROR will be executed each time a program
     error occurs. If placed elsewhere in the
     program, the statement will be executed only
     for errors which occur during the execution of
     statements following the execution of the ON
     ERROR statement.

3.   ON ERROR is reset by RUN and SCRatch
     instructions.

Example:

```
 60 INPUT A,B
 80 PRINT A*B
100 ON ERROR GOTO 300
120 INPUT C,D
140 PRINT C/D
160 GOTO 60
300 PRINT "A non-fatal error has occurred"
320 GOTO 120
```

In this example, any error which occurs before line
100 has been executed for the first time will be
dealt with by the standard system error handling
procedure. Any trappable error which occurs after
line 100 has been executed will cause program
execution to continue with statement line 300.

instruction: ON ESCape

     format: [Ln] ON ESC STOP

             [Ln] ON ESC GOTO nl

             [Ln] ON ESC GOSUB nl

where:

   Ln   is an optional line number.

   nl   is the line number of the statement to which control is transferred.

The ON ESC instruction causes program control to be transferred as specified (STOP, GOTO, or GOSUB) when the ESCape key is depressed during program execution.

Notes:

1.   If ON ESC is written at the beginning of a program, the instruction specified with ON ESC will be executed each time the ESCape key is depressed. If placed elsewhere in the program, the statement will be executed only when the ESCape key is depressed following the execution of the ON ESC statement.

2.   ON ESC is reset by RUN and SCRatch instructions.

Example:

```
>>LIST

    10  INPUT A,B,C
    15  IF A = -1 THEN 60
    20  LET D=A+B+C
    30  ON ESC GOTO 10
    40  PRINT D
    50  GOTO 50
    60  END

>>RUN
? 10,15,20
45                  (escape must be pressed here to
                    get out of the loop statement
                    50.)

? 70,10,5
85
? -1,0,0
***60 END***
```

instruction: RENAME

format: [Ln] RENAME svar-1, svar-2

where:

Ln   is an optional line number.

svar-1   is a string literal or string
            variable file reference to the
            existing file.

svar   is a string literal or string
            variable file reference to the
            new file.

The RENAME instruction is used to give a new file
name to a file already in the file directory.

Example:

    RENAME "OLDFILE", "NEWFILE"

This command will change the name of the file (on
the current disk drive) from OLDFILE to NEWFILE.

221

instruction: SET system parameter

format: SET aexp-1, aexp-2

where:

aexpl-2   are arithmetic expressions,
variables, or constants.

aexp-1   is the system parameter
number.

aexp-2   is the value to be
assigned to the
parameter.

The SET instruction is used to change the value of
a given system parameter.

Notes:

1.   Refer to the SYS instruction for a list of all
user accessible system parameters.

2.   System parameter number Ø is the width of a
page in characters and may assume any value
between Ø and 32767. The default value for
this parameter is 8Ø characters which
corresponds to a standard 8 1/2 inch page.

A special use of the set command is:

SET Ø, -1

which inhibits the automatic carriage RETURN-
line feed at the end of a line when page width
is reached. Refer to the discussion of the
PRINT instruction as it pertains to file
output.

Using SET Ø,-1 to disable page width checking
is especially useful for graphics output to
devices such as the Cromemco 3355 printer.

3.   System parameter 1 is the tab field width and
may assume any value between 1 and 32767.
This corresponds to the width of the field
which is output by a comma (,) in a PRINT
instruction. If the tab field width is set to
a Ø or -1, no tabbing takes place (i.e.,

commas are treated as semicolons). The default value for this parameter is 20 characters. This default value may be changed, see the Appendix, Areas of User Interest.

4. For disk BASIC only, a System parameter has been added to facilitate timed input. To start the timer, the SET 5 instruction is given, with aexp-2 equal to approximately ten (10) times the number of seconds of delay desired. Halve aexp-2 if using a 2MHz clock. For example:

SET 5,50

will allow about 5 seconds for a complete user response.

When the next INPUT instruction is encountered, BASIC will issue an ERROR 210 -- INPUT TIMEOUT if the user does not respond with a <u>complete</u> input within the allotted time.

Once set, the time used to respond to all subsequent INPUT instructions is added together until it excedes the specified value, and then an error message is generated. It is therefore necessary to give the SET instruction prior to each timed INPUT. When timed INPUT is no longer desired, it may be de-activated by coding:

SET 5,0

The ON ERROR statement can be used to trap the timeout error. The programmer may find out how much time was used by coding SYS(5) to find the time remaining.

function: SYStem

format: SYS (aexp)

where:

aexp   is an arithmetic expression,
            variable, or constant.

The SYS function provides system information based
on the value of the argument aexp.

Notes:

1. See the SET instruction for methods of
    changing the default values of these system
    parameters.

Argument Value	System Parameter Returned
Ø	page width (number of characters)
1	tab field width (number of characters)
2	character last printed (i.e., current character)
3	last runtime error number (for disk BASIC) last runtime error two-letter ASCII code (for stand-alone BASIC)
4	current print column number
5	current input timer value

Example:

```
100 ON ERROR GOTO 700
    .
    .
140 OPEN \1\ "TESTFILE.DAT"
    .
    .
700 IF SYS(3)=128 THEN CREATE"TESTFILE.DAT": GOTO 140
    .
    .
```

In this example, if the file TESTFILE.DAT did not originally exist on the disk, the error handler at line 700 would CREATE the file. Lines 710, 720 etc. could be written to test for other values of SYS(3), and take appropriate action(s).

instruction: INPut

format: INP (m)

where:
m   is an I/O port.

The INP instruction is used to read the contents of
INPut port (m).

Note:

1.   The value of m must be in the range 0≤m≤255.

Example:

```
10 NOESC : INTEGER K
20 IF BINAND(INP(0),%0040%) = 0 THEN GOTO 20
30 K = INP(1)
40 K = BINAND(K,%007F%)
60 IF K = %001B% THEN PRINT : GOTO 100
70 PRINT CHR$(K);
80 IF K = 13 THEN PRINT
90 GOTO 20
100 ESC
110 END
```


When a character is typed at the console, this
program will echo it back to the console CRT. A
carriage RETURN will be echoed as a carriage RETURN
followed by a line feed, and an ESCape will
terminate the program.

The NO ESCape instruction at line 10 is necessary
to disable BASIC's continuous polling of input port
0.   While a program is being executed, basic
ignores most characters typed at the console.
BASIC continually polls the UART status port 0 and
if a character is ready, reads it from port 1.   If
this character is not an ESCape character, it is
ignored by BASIC.   If it is an ESCape character,
program execution is terminated.

Upon reading the UART status, the status bit is
reset.   It is necessary to disable BASIC's polling
of the status port so that this user program will
be able to determine if a character has been typed.
If this were not done, BASIC would, by reading the

status port, reset the character ready bit, thereby
preventing the user program from determining that a
character had been typed.

Line 60 replaces the ESCape key function by testing
the character typed by the user. If it is an
ESCape, program execution is terminated.

Statement 20 is a loop which continually polls the
UART status ready port to determine if a character
has been typed. The Receiver Data Available flag
is bit 6 (refer to the 4FDC manual). The contents
of input port 0 is ANDed with 40 hex to determine
if this bit is a one or a zero. If it is a zero,
no data is available, and control remains on line
20, polling the input port again.

If the bit has been set (=1) it indicates that data
is available, and statement 30 reads the data from
input port 1 and assigns this data to the variable
K. Line 40 strips the parity bit off of the input
data (the ASCII character set requires only the
seven bits, 0 through 6) and as stated before, line
60 terminates program execution upon determining
that an ESCape character (1B hex) has been input.

The rest of the program prints the ASCII character
whose value was input, outputs a line feed if a
carriage RETURN was input, and transfers control to
line 20.

When an ESCape is detected, control is transferred
to line 100 which re-enables the operation of the
ESCape key and terminates program execution.

instruction: OUTput

format: OUT m, b

where:

m   is an I/O port

b   is a byte value

The OUT instruction is used to OUTput data byte (b)
to OUTput port (m).

Notes:

1.   The value of m must be in the range $0 \le m \le 255$.

2.   The value of b must be in the range $0 \le b \le 255$.

Example:

>>OUT 1,75

will display the character K (ASCII 75) on the
console terminal (OUTput port 1).

instruction: PEEK

format: PEEK (m)

where:

m   is a memory location.

The PEEK instruction returns the contents of memory location (m).

Note:

1. If m is decimal, it must be in the range: $0 \leq m \leq 32767$. Hexadecimal numbers may be used to access locations in the range: $0 \leq h \leq \%FFFF\%$.

Example:

PRINT PEEK (5)

will print the contents of memory location 5.

instruction: POKE

format: POKE m,b

where:

m   is a memory location.

b   is a byte value.

The POKE instruction puts the byte value (b) into memory location m.

Notes:

1.   If m is decimal, it must be in the range: $0 \leq m \leq 32767$. Hexadecimal numbers may be used to access locations in the range: $0 \leq h \leq \%FFFF\%$.

2.   The value of b must be in the range $0 \leq b \leq 255$.

Example:

POKE 1000,255

will place the value 255 into memory location 1000.

function: USeR program

format: USR (m,Pl,...,Pn)

where:

m   is the address of the assembly
            language routine.

Pl-n   are parameters which are
            converted to 16 bit integers.

The USR function makes it possible to call an
assembly language subroutine from a BASIC program.

Notes:

1.   The USR function always requires the user to
     specify one parameter in addition to the
     address, even if it is a dummy parameter. For
     example, USR(Ø,1) is correct, while USR(Ø)
     will result in a syntax error.

2.   If m is decimal, it must be in the range:
     Ø≤m≤32767. Hexadecimal numbers may be used to
     access locations in the range: Ø≤h≤%FFFF%.

3.   USR is a function and must take the form of a
     function when it is used:

          >>1ØØ A = USR (X,Y)

4.   When the user routine gains control (at the
     address specified in the USR function call),
     the following conventions apply:

     a)   Register A contains the number (n) of
          parameters in the function call.

     b)   Register pair HL contains the return
          address to BASIC. The user routine may
          re-enter BASIC by coding the following
          assembly language instruction:

               JP (HL)

     c)   The parameters were placed in order
          (i.e., Pl was PUSHed first, followed by
          P2, P3,..., Pn) on the CPU stack and may
          be recovered via POP instructions.

d) If and only if n parameters (n is the contents of register A, as above) are POPped off the stack, and the stack pointer is not otherwise changed, BASIC may be re-entered via the following instruction:

    RET

e) The routine may return a 16-bit value to be assigned to the function by placing the value in DE register pair before re-entering BASIC.

f) Aside from the restrictions noted above, all registers may be used in any way by the user routine.

Example:

The following routine, when POKEd into position by the BASIC program starting at line 100, will allow the BASIC user to perform CDOS system calls. The USR function must be used to access this routine. The first parameter of the USR function is the starting address of the assembly language routine (in this case, 103 Hex). The second and third parameters are the values which are to be loaded into the DE and BC register pairs before the system call. Upon returning from the routine, the entire USR function takes on the value of the A register. This value may be accessed by assigning the USR function to a variable. Notice that the C portion of the BC register pair always contains the CDOS system call number.

```
 89 REM DATA statements 100 & 110 are the machine
 90 REM code equivalent of the following assembly
 91 REM language routine:
 92 REM
 93 REM pop de
 94 REM pop bc
 95 REM call 5
 96 REM ld d,0
 97 REM ld e,a
 98 REM ret
 99 REM
100 DATA %00D1%,%00C1%,%00CD%,%00005%,%00000%
110 DATA %00016%,%00000%,%0005F%,%00C9%
120 INTEGER A,I
130   FOR I = %0103% TO %010B%
140   READ A
150   POKE I,A
160   NEXT I
170 REM
1000 REM Following BASIC program demonstrates
1100 REM the CDOS system call which positions the
1200 REM CRT cursor.
1300 REM
1400 SET 0,-1 : REM disable page width checking.
1500 REM clear screen and home cursor.
1600 A = USR(%0103%,0,142)
1700   FOR I = 1 TO 24
1800   GOSUB 4400
1900   NEXT I
2000 REM
2100 REM Clear and home cursor.
2200 A = USR(%0103%,0,142)
2300   FOR I=24 TO 1 STEP -1
2400   GOSUB 4400
2500   NEXT I
2600 GOTO 1600
2700 REM
2800 REM
4000 REM subroutine to compute diagonal column
4100 REM given a row number, position cursor
4200 REM at the given location, and print the
4300 REM index number there.
4400 J = I*3 : REM compute column number
4500 REM Move column number into high byte
4600 REM and row number into low byte
4700 K = 256*J+I
4800 A = USR(%0103%,K,142)
4900 PRINT I;
5000 RETURN
5100 END
```

GLOSSARY

[ ]

Square brackets are used to indicate an optional quantity. The item enclosed in square brackets may be used, in the position indicated, at the user's discretion.

Argument

An argument is an independent variable, constant, or expression used with a BASIC instruction whose value can be specified by the user to instruct BASIC to perform a certain task. For example, in the instruction:

        PRINT A, 3, C+7

A, 3, and C+7 are arguments to the BASIC instruction PRINT.

ASCII

This acronym stands for American Standards Code for Information Interchange. It is an industry standard used to assign numerical codes (0 through 127) to 128 characters used as letters, numbers, arithmetic operators, various symbols, and control characters. The ASC(X) function will return the ASCII equivalent of any argument. A table of ASCII codes is provided for reference in the Appendix.

BASIC Word

A BASIC word, commonly called an instruction, is an alphanumeric set of characters which briefly describes the operation to be performed by the computer. Some examples of BASIC words are:

    LIST
    PRINT
    ON ERR
    LEN
    STOP
    RND

Binary Code

Binary code is defined as a code where every code element is either a 0 or a 1. Computer instructions and data for most microcomputers consist of unique, 8 bit binary codes.

Command

A command in BASIC is an instruction to the computer which specifies an operation to be performed. In contrast to a BASIC statement (see the STATEMENT definition), commands are executed immediately. Commands are used primarily to manipulate or execute a program once the program has been entered. Commands have no line number preceding them.

A powerful feature of Cromemco 16K BASIC is the ability to use most commands as statements. As such, they may be given line numbers and included in the body of the program for execution while the program is running.

Control Character

A control character is a non-printing ASCII character which is (usually) used to transmit control signals between a peripheral device and the computer. For example, a CTRL-P entered from the console will cause the system printer to echo all information which is displayed on the console.

Current Program

The current program is any program with which the user is currently interacting. When 16K BASIC is entered, no program is current. Should the user enter text to create a new program, this program becomes the current program. If the user calls a saved program from system memory, that program becomes the current program. If the user edits a program, it remains the current program in its edited form.

Data

The term is used in two ways. Strictly speaking, any information contained within memory or control logic is binary data. Whether this data becomes alphanumeric characters or control information depends upon the program in use.

In the other sense, data is used to refer to numerical or string information. In BASIC, this numerical or string information is listed in a file or DATA statement.

Default

With certain BASIC instructions, an argument may be optionally added to control a certain function. If no argument is given, the instruction defaults or reverts to a value already programmed into the BASIC interpreter. For example, the default values of the arguments for the command:

    RENUMBER

are 10,10 in Cromemco BASIC. This default value
for RENUMBER will produce automatic line
renumbering starting with line 10 and numbering
consecutive lines by increments of 10, (e.g., 10,
20, 30, 40...). To change this default value, the
BASIC word must be followed by an argument. For
example, the command:

        RENUMBER 5,5

will provide automatic line renumbering starting
with line 5 and continuing by increments of 5
(e.g., 5, 10, 15...).

Disk Storage

A disk is a computer memory device which is used to
store information. Disks are typically used in
place of main memory when large amounts of
information must be stored. A disk is similar in
appearance to a phonograph record. Most
microcomputer systems currently offer disk storage
capabilities through either large or mini floppy
disks. The floppy and mini floppy terms refer to
the two different sizes (8 inch and 5 inch
respectively) of the flexible plastic disks used
with the disk assemblies.

Expression

An expression is defined as any combination of
variables, constants and operators which is
evaluated as a single value or logical condition.
For instance, in the statement:

        10 LET A = (B*C) + (A*D)

the (B*C) + (A*D) operation, the value of which is
assigned to variable A, is interpreted as an
expression. In the statement:

        10 IF A = B THEN GOTO 250

the logical comparison A = B is called an
expression and is evaluated to True (=1) or False
(=0).

238

File or Data File

A File defines a group of related information. This information is addressed by means of a File Reference and usually resides on a floppy diskette.

File Name

This is a one to eight character label which is used to refer to a File. Several Files may have the same File Name. These Files may be uniquely identified by the use of a Disk Specifier and/or a File Name Extension.

Firmware

Firmware is the middle ground between hardware and software. This term is generally applied to specific software instructions that have been burned in or programmed into Read Only Memory (ROM).

Floating Point Mode

Floating point mode refers to a method of computer calculation in which the computer keeps track of the decimal point in each number. In 16K BASIC, three formats are used to define variables: Integer, Long Floating Point, and Short Floating Point. In the Long Floating Point mode, numerical values are allowed up to 14 digits. In the Short Floating Point mode, numerical values are limited to 6 digits. The default value in Cromemco BASIC is the Long Floating Point (LFP) mode.

Hardware

In comparison to firmware and software, hardware
represents the actual metal (or hard) elements of a
computer system.   Items such as printers,
terminals, and the computer itself are considered
to be hardware.

Integer

An integer is defined as a whole numbeY, positive
or negative. The following numbers are examples of
integers and non-integers:

INTEGERS	NON-INTEGERS
3	3.14159
1Ø	.66666
-5	2/3

Integer Mode

Integer mode is a format used to define variables
in which one or all variables within a given
program are set to integer values only.

Interactive

An interactive device is one used to achieve direct
person to computer communication, and vice versa.
The teletype and CRT terminals are the best known
examples of interactive terminals, although many
variations are possible.

I/O (Input/Output)

The I/O initials stand for Input and Output. I/O is the transfer of data between the computer system and an external device. Devices such as CRT (Cathode Ray Tube) terminals, TTY (teletypewriter) terminals, and disk drives are examples of devices that accept the input data from the user, another peripheral device, or from the computer memory, and that output data to the computer.

Line Number

All lines in BASIC begin with a line or statement number. For example:

        10 PRINT A, B

includes the statement number 10. Line numbers can be assigned manually or through the AUTOL command and may be any integer from 1 through 99999. All BASIC lines have a unique number which is used to access lines which require modification or deletion from the program.

Matrix

A matrix is an array of numeric variables in a prescribed form. For example, the array:

            3   2   0

            1   4   6

           -3   4   5

is a matrix with three rows and three columns. A matrix with m rows and n columns is written:

        a11 a12 a13 ...a1n

        a21 a22 a23 ...a2n
        .
        .
        .
        am1 am2 am3 ...amn

241

The individual entries in the matrix are called
elements or cells. For example the quantity aij in
the above matrix is the element in row i and column
j. Subscripts used to indicate elements always
denote the row first and the column second.
Cromemco BASIC permits the user to define one, two,
or three dimensional matrices. A two (i.e., Mij)
or three (i.e., Mijk) dimensional matrix is
commonly called a table. A one dimensional matrix,
a matrix with n columns but only one row, is
commonly called a list. For example, the matrix:

        3, -1, 5, -8

is a list (or a matrix) with one row and four
columns.

Memory

The computer memory is used to store information,
including programs and data, for future use.
Microcomputers typically use semiconductor
memories, of which the two most common types are
random-access memory (RAM) and read-only memory
(ROM). From a hardware perspective, memory
consists of an array of bistable, individually
addressable elements each of which represents a
single binary digit. Information can be stored
either in main memory, which commonly consists of
RAM or ROM, or external storage devices, which
include disks, magnetic tape, and magnetic drums.

Peripheral Device

Peripheral devices are units which are used in
conjunction with a computer but which are external
to the computer. Peripherals refer to devices such
as printers, plotters, terminals, disk storage
devices, etc., which can be connected to the
computer. The computer is assumed to be the
central unit and peripherals are merely support
devices.

Program

A computer program is a set of instructions arranged into statement lines. The instructions are used to instruct the computer to perform specified operations in a certain order. Programs are designed and written to solve a wide range of problems and are used in applications as varied as process control, data reduction, telephone systems, mathematical analysis, games, and stock market transactions.

PROM

This acronym stands for Programmable Read Only Memory. PROMs consist of an array of memory cells that can be fixed in certain patterns by the application of higher than normal voltages. These memories are said to be non-volatile; that is, when power is withdrawn the programmed pattern remains.

Recently, EPROMs, or Erasable PROMs, have appeared and have found industry wide usage. EPROMs may be erased by exposure to ultraviolet light, and then re-programmed. The Cromemco Bytesaver II is designed to program EPROMs.

Protocol

Protocol is a set of conventions on the format and content of messages to be exchanged between two logical devices. Most often, differences in timing account for failure of devices to communicate. For example, a certain signal might, of necessity, be present to enable an I/O request to a microprocessor's protocol. To match a computer to a terminal, one must know the mutual handshake protocol.

RAM

RAM stands for Random Access Memory, or read-write
memory. In contrast to PROMs, read-write memory
can be changed as well as being read. Some RAMs
(known as dynamic) retain data for only a fraction
of a second and must be refreshed constantly to
retain data. All RAM is volatile and must have
power applied to retain data patterns.

ROM

A ROM is a Read Only Memory device that is used for
storing fixed information. This information is
burned in, or programmed, at specific locations
when the ROM is manufactured. A ROM cannot be
written into during operation. Any ROM that can
later be altered is a Programmable Read Only Memory
(see PROM). ROM family memories, once burned,
retain their data regardless of power
contingencies.

Sector

A Sector is a subdivision of a track. A track on a
large disk is divided into 26 sectors while a small
disk track is divided into 18 sectors. Sectors are
numbered starting from number one and each sector
holds one record or 128 (80H) bytes.

Software

Software is a term used to refer to the programs,
languages and procedures used in a computer. For
example, the 16K BASIC interpreter as well as any
BASIC programs are identified as software.

Statement or Statement Line

A statement in BASIC is an instruction or series of
instructions to the computer. A statement is
defined as one line in a BASIC program which is
preceded by a line number. For example:

        100 A = B*C

is defined as a statement. Typically, a statement
can contain a maximum of 132 characters.

A powerful feature of Cromemco 16K BASIC is the
ability to use most statements as commands. As
such, they may be used without line numbers and
executed immediately. This is very useful for
debugging programs.

Cromemco 16K BASIC also allows more than one
instruction on a single statement line as long as
adjacent instructions are separated by a colon (:).

String Literal

A string literal (or string) is a sequence of
alphanumeric characters, spaces, and special
characters. In 16K BASIC, string literals must be
enclosed within quotation marks. Examples of valid
string literals include:

        "CROMEMCO 16K BASIC"

        "12345"

        "THIS PROGRAM PRINTS SQUARE ROOTS"

The statement:

        100 PRINT "CROMEMCO 16K BASIC"

will output the string

        CROMEMCO 16K BASIC

String Variable

A string variable is a variable which may assume
the value of a string literal.

Track

A Track is a physically defined circular path which
is concentric with the hole in the center of a
disk.   It   is   defined   by   its distance from the
center of the disk.   With the read/write head of
the disk drive located on a given track, data may
be read from or written to that track.   A large
disk has 77 tracks, while a small disk has 40.

User Area

The User Area is the BASIC workspace in which a
program can be written, edited, and run.   The LIST
command displays the contents of the User Area.

Variable

A variable is a quantity that can assume any one of
a given set of values.   In 16K BASIC, variables are
defined by a single letter (A through Z) or a
single letter followed by one digit (0 through 9).
Examples of legal variable names include:

    A

    A1

    C

    C0

Variables   represent   numeric   values.   In   the
statement:

    20 A = 8 + 2

A is the variable and 8+2 or 10 is the value
assigned to A.   A new value can be assigned to A at
a subsequent point in the program.

BASIC ERROR MESSAGES

20.1   Fatal Errors

Stand Alone BASIC Error Code	Disk BASIC Error Number	Message	Meaning

SY	1	Syntax	This error message covers a number of errors which can occur when the user is entering (typing in) a program. For example:
			 Unmatched parentheses:
			    A=(B*(C)
			 Misspelled words:
			    PIRNT A
			 Wrong data type: A$=3*A
			 Bad punctuation:
			    PRINT A(7;2)
			Because there is only one message for all these errors, a dollar sign is printed under the line in error at a position approximately indicating the position of the error.

US	2	Using Syntax	The format string for a PRINT USING instruction is in error. For example: PRINT USING "#.##!!!", 3.2E9 (only 3 exclamation marks; 4 required)

#A	3	Number of Arguments	A function call requires a different number of arguments than the number passed to it. For example:
			`DEF FNA(X,Y)=X+Y`
			`PRINT FNA(J)`

*	4	No Disk in System	In a stand alone system, a request for a disk-only function was made. `e.g., OPEN\1\"LP"` (presumably the user meant "$LP")
IL	5	Illegal Statement	(1) This can be caused by entering a line with a syntax error and then RUNning the program without correcting the line. (2) In certain systems, certain statements can be declared invalid. For example, POKE might be illegal in a time-sharing system.

PS	6	Print Item Size	An attempt was made to print a single item which required more characters than the current page width. e.g.,
			`SET 0,10`
			`PRINT "LOTS OF CHARACTERS"`

#G	7	Too Many GOSUBs	Subroutines are nested within subroutines to a depth which exceeds that allowed by BASIC.
#(	8	EXP Too Complex	Too many levels of expressions, too many parentheses or function references.

RT	9	Return, No GOSUB Active	The program has no place to return. This can be caused by deleting a line with a GOSUB statement and then encountering its RETURN statement.
NX	10	NEXT Without FOR	FOR and NEXT statements must be paired. This error may occur if a line containing a FOR statement is deleted.
FU	12	Function not Defined	A user function is referenced by the program but has not been defined. If the line containing the DEF FNs(X) is deleted, the function is no longer defined.
DV	13	DIMension Statement Error	Invalid argument(s) in the DIMension statement. For example: a negative number, DIM A (-20); too many subscripts, DIM B (5,5,5,5); too large an integer, > 16382.
L#	14	GOTO/GOSUB Undefined Line Number	A GOTO or GOSUB statement refers to a line that does not exist.
SS	15	Subscript Values	The values assumed by subscripts must be less than those in the DIMension statement.
#S	16	Number of Subscripts	The number of subscripts associated with a variable must match the number of subscripts in the DIMension statement.
EL	101	End of Statement/ End of Line	This is an internal BASIC error - please document and mail to Cromemco, Customer Service Dept.

| SZ | 102 | Array or String Space Overflow | There is not enough memory to store the array or string. |

20.2   <u>User</u> <u>Trappable</u> <u>(Non-Fatal)</u> <u>Errors</u>

Stand Alone BASIC Error Code	Disk BASIC Error Number	Message	Meaning
NF	128	File not Fnd	File not found on disk (file not in directory) or the device name is not in the device directory list.
FN	129	Filename	A disk file name was used in a Stand Alone System or an illegal file name was passed.
CM	130	Invalid Cmd for Device	A command was given to a device which that device was incapable of performing. For example: a read command given to a line printer.
FO	131	File Already Open	An OPEN command was given to a file which was already open.
NO	132	File Not Open	A read or write was attempted using a file which had not been OPENed.
F#	133	File Number	The file number requested was outside the allowable range. The file number must be greater than 0 and less than or equal to the maximum channel number.
OP	134	Cannot Open File	A message from the device driver (or CDOS). (A non-zero value returned on OPEN.)

FS	135	No File Space	All files in use. The system must have one unused channel to do a LIST, ENTER, SAVE, or LOAD.
			CDOS only - no more space on disk (or there are 64 directory entries).
*	136	File Mode Error	A READ was attempted from a write only file or vice versa.
*	137	File Already Exists	An attempt was made to CREATE a file that already exists.
*	138	File Read: No Data	End of file read, or, for random access only, an attempt to read a portion of the file which had not been written.
*	139	File Write	A message from CDOS — an attempt was made to write to a write protected disk or an error occurred while writing to the disk.
*	140	File Position	An attempt was made to read a negative file record or record larger than 240K bytes.
NC	141	No Channel Available	All I/O channels in use. (The maximum number of channels is system dependent.)
HX	200	Invalid Hex Number	Hexadecimal numbers must contain only the characters 0 through 9 and A through F.
BO	201	Integer Overflow	A value greater than 32767 was assigned to an integer variable.

FA	202	Function Arg Value	A function was called using an illegal argument. For example: SQR (-2).
IN	203	Invalid Input	An attempt was made to INPUT non-numeric data into a numeric variable.
IN	204	Input	An attempt was made to INPUT more items that were called for in the INPUT instruction.
DM	205	Not Dimensioned	A reference was made to a subscripted variable which had not been dimensioned.
ND	206	No Data Statement	An attempt was made to READ past the end of the DATA supplied. Either there was a READ with no DATA statement or there were not as many items in the DATA statement as in the READ list.
DT	207	Data Type Mismatch	An attempt was made to READ a numeric value to a string variable or vice versa. For example: 10 DATA 5 20 READ A$
E#	208	Number Size	An attempt was made to assign a value outside of the range $9.99E+62$ to $9.99E-65$ to a variable.
LL	209	Line Length	A line longer that 132 characters was entered.
*	210	Input Timeout	See the SET instruction for information about this error.

OV	250	Overflow/ Underflow	A floating point operation produced a number outside of the range 9.99E+62 to 9.99E-65. For example: A=1/0. Or, Integer arithmetic caused results outside of the range -32768 to 32767.

Note:

The character * in the Stand Alone BASIC Error Code column indicates that the error message is not applicable.

254

ASCII CHARACTER CODES

DEC.	HEX	CHAR.		DEC.	HEX	CHAR.	DEC.	HEX	CHAR.
000	00	NUL	(CTRL-@)	043	2B	+	086	56	V
001	01	SOH	(CTRL-A)	044	2C	,	087	57	W
002	02	STX	(CTRL-B)	045	2D	-	088	58	X
003	03	ETX	(CTRL-C)	046	2E	.	089	59	Y
004	04	EOT	(CTRL-D)	047	2F	/	090	5A	Z
005	05	ENQ	(CTRL-E)	048	30	0	091	5B	[
006	06	ACK	(CTRL-F)	049	31	1	092	5C	\
007	07	BEL	(CTRL-G)	050	32	2	093	5D	]
008	08	BS		051	33	3	094	5E	^
009	09	HT		052	34	4	095	5F	<
010	0A	LF		053	35	5	096	60	'
011	0B	VT		054	36	6	097	61	a
012	0C	FF		055	37	7	098	62	b
013	0D	CR		056	38	8	099	63	c
014	0E	SO	(CTRL-N)	057	39	9	100	64	d
015	0F	SI	(CTRL-O)	058	3A	:	101	65	e
016	10	DLE	(CTRL-P)	059	3B	;	102	66	f
017	11	DC1	(CTRL-Q)	060	3C	<	103	67	g
018	12	DC2	(CTRL-R)	061	3D	=	104	68	h
019	13	DC3	(CTRL-S)	062	3E	>	105	69	i
020	14	DC4	(CTRL-T)	063	3F	?	106	6A	j
021	15	NAK	(CTRL-U)	064	40	@	107	6B	k
022	16	SYN	(CTRL-V)	065	41	A	108	6C	l
023	17	ETB	(CTRL-W)	066	42	B	109	6D	m
024	18	CAN	(CTRL-X)	067	43	C	110	6E	n
025	19	EM	(CTRL-Y)	068	44	D	111	6F	o
026	1A	SUB	(CTRL-Z)	069	45	E	112	70	p
027	1B	ESC	(CTRL-[)	070	46	F	113	71	q
028	1C	FS	(CTRL-\)	071	47	G	114	72	r
029	1D	GS	(CTRL-])	072	48	H	115	73	s
030	1E	RS	(CTRL-^)	073	49	I	116	74	t
031	1F	US	(CTRL-_)	074	4A	J	117	75	u
032	20	(SPACE)		075	4B	K	118	76	v
033	21	!		076	4C	L	119	77	w
034	22	"		077	4D	M	120	78	x
035	23	#		078	4E	N	121	79	y
036	24	$		079	4F	O	122	7A	z
037	25	%		080	50	P	123	7B	{
038	26	&		081	51	Q	124	7C		
039	27	'		082	52	R	125	7D	}
040	28	(		083	53	S	126	7E	~
041	29	)		084	54	T	127	7F	DEL
042	2A	*		085	55	U			

LF=Line Feed FF=Form Feed CR=Carriage Return DEL=Rubout
ESC=ESCAPE

OVERLAY STRUCTURE

These programs demonstrate the ability of
Cromemco's 16K Extended BASIC to overlay or have
one program overlay (part of) itself with another.
This feature is very useful for running large
programs on smaller systems.

Space is most efficiently used if the same line
numbers are used in all programs which are to be
overlayed. In the following example, program ONE
is run and calls in program TWO.

        PROGRAM "ONE"

            1 GOTO 10
            5 ENTER "TWO"
           10 DATA 1,2,3,4,5,6,7,8,9
           20 READ A,B,C,D,E,F,G,H,I
           30 GOTO 5

        PROGRAM "TWO"

           10 X=A+B+C+D
           20 Y=E+F+G+H+I
           30 Z=X+Y
           40 PRINT X,Y,Z
           50 END

        >>ENTER "ONE"

        >>RUN
        10   35   45
        50 END

When program ONE is entered and RUN, line 1 passes
control to line 10. Then at line 20, the DATA from
line 10 is READ into variables A through I.
Control is then passed to line 5 which calls in the
second program. Program TWO overlays lines 10, 20,
and 30. The line which follows line 5 is line 10.
Now, however, line 10 is a line from program TWO.
The rest of program TWO is executed with the
results being printed out by line 40.

256

AUTOMATIC STARTUP AND

PROGRAM EXECUTION FROM CDOS

A very powerful feature of the Cromemco Disk
Operating System (CDOS) is the ability to enter
directly into an application program when powering
up the computer. This is especially useful for the
inexperienced user as there is no need to deal with
any of the commands which are used to load and
execute a program.

If, for example, the user wants to execute the
BASIC program 'START.SAV' automatically when CDOS
is entered, the following steps should be followed:

1.   Make sure that there is a copy of the
      batch command file '@.COM' on DISK A.

2.   Save the BASIC program you want RUN in a
      file (in this example we are using
      'START.SAV'). The program must be SAVEd
      (not LISTed) in order for this to work!
      Our program for this example is:

           100 REM THIS IS MY
           105 REM APPLICATION PROGRAM!
           110 A=5
           120 B=10
           130 PRINT "THE ANSWER IS: "; A*B
           140 END

3.   Using the editor, create a file named
      'STARTUP.CMD' on a disk A. Note that
      this must be named 'STARTUP.CMD' as this
      is the file name that CDOS looks for. In
      this example, the command file should
      contain the line:

           BASIC START.SAV

      Then, when CDOS is entered, the batch
      command will call BASIC which will RUN
      the saved program 'START.SAV.

4.   When the computer is turned on and CDOS
      is entered (you have to hit the carriage
      return several times), our example will
      output the following:

            CROMEMCO CDOS VERSION XX.XX

            A.@ STARTUP
            BATCH VERSION XX.XX

            A.BASIC START.SAV

            CDOS 16K BASIC, VERSION X.XX

            THE ANSWER IS: 50

            140 END

5.   If the first program line is a NOESC or
      ONESC, the user can be prevented from
      ever leaving the running BASIC program.

MIXED MODE ARITHMETIC

There are three types of numbers in BASIC, Integer
Short Floating Point (SHORT), and Long Floating
Point (LONG), (hexadecimal constants are treated as
integers). Any arithmetic computation, assignment,
INPUT, or READ can be performed using any one or
more of these types of numbers.

An assignment, READ, or INPUT will automatically
convert the number to the type of the receiving
variable. This is the variable which is on the
left of the equal sign in an assignment
instruction, and in the data list in the READ and
INPUT instructions.

In general, numbers are converted to other types
freely as needed. For example:

        SIN (30)   works, even though the SIN function
                    must have a LONG argument. The
                    Integer 30 is converted to a Long
                    Floating Point number.

        SYS (2.6)   works, even though the SYS function
                    must have an Integer argument. The
                    Long (or Short) Floating Point
                    number 2.6 is converted to an
                    integer (rounding to 3.0).

Most problems will occur with mixed mode arithmetic
involving Integer numbers. Remember that all
constants without a decimal point and with a value
less than 10,000 are stored as Integers.

Examples:

        >>SHORT S
        >>INTEGER I
        >>S = 6 : I = 1
        >>S = I / 3 * S

This example will assign a value of 0 to S. This
is because I/3 is evaluated first (rules of

259

precedence, arithmetic operators). Because both I
and 3 are Integers, I/3 is evaluated as a 0 using
Integer arithmetic. Zero times anything, no matter
what type, is still 0.

```
>>SHORT S
>>INTEGER I
>>S = 6 : I = 1
>>S = S * I / 3
```

This example will assign a value of 2 to S. This
time S*I is evaluated first, and because this is
mixed mode arithmetic, the shorter form is
converted to the longer form (I's value is
converted to SHORT). Then S*I is equivalent to
6.0*1.0 or 6.0. We are left with 6.0/3, mixed mode
again.

The Integer 3 (the shorter type) is converted to
Short Floating Point (the longer type) and the
division is performed. S is assigned a value of
6.0/3.0 or 2.0.

```
>>INTEGER S
>>SHORT S
>>LONG L
>>I = 1 : S = 3 : L = 1
```

If at this point we give the command:

```
>>L = L/S
```

L will be assigned a value of 0.33333333333333
because L/S is evaluated as Long Floating Point
(the longer type).

If instead we have given the command:

```
>>L = I/S
```

L would have been assigned the value of
0.333333000000000 because I/S is evaluated as Short
Floating Point (the longer type, but still only 6
digits of accuracy) and then assigned to a Long
Floating Point variable (L) with 14 digits of
accuracy.

Conversion of both types of floating point numbers
to Integer numbers and vice versa does take time.
Also Arithmetic, indexing, and subscripting with
floating point numbers is much more time consuming

than it is using Integers.

Time can be saved by using Integer numbers wherever possible. Where it is not possible, care in precedence ordering can result in significant time savings. If L is type LONG and I is type INTEGER, the first of the following commands will execute faster than the second one.

```
>>L = I*I*I*I*L
>>L = L*I*I*I*I
```

This is because, until the last multiplication, the first example is using Integer arithmetic. The second example uses Long Floating Point arithmetic from the start, because the LONG variable is at the left and this is where evaluation of this expression begins.

PATCH SPACE

Disk Version: The only space available in the disk
version of 16K BASIC for user patches is at
hexadecimal locations 425 to 4FF and 103 to 1FF.
The user can put and SAVE (using CDOS) a patched
ROUTINE at these locations.

CROMEMCO 16K EXTENDED BASIC

AREAS OF USER INTEREST

Notes:

1. XX stands for: 04 if you are using CDOS
                   80 if you are using SAB

2. CDOS is the Cromemco Disk Operating System
   SAB is the BASIC Stand Alone Operating System

3. Many of these values may be changed by the SET
   instruction. If necessary, and with care,
   they may be changed with the POKE instruction.
   Refer to the Appendix section on Changing the
   number of I/O Channels for an example.

Hex Address	No. of Bytes	Version	Description
XX00	3	CDOS	A jump (JP) to the routine where BASIC uses the address of the bottom of CDOS to establish the top of user program space.
		SAB	A jump (JP) to the routine which uses repeated RETURN characters to initialize the baud rate. After baud rate is established, BASIC automatically "sizes" RAM memory (starting at location 0) to establish top of user program space.
XX03	3	Both	A jump (JP) to a point in BASIC equivalent to a request to enter a new program line. Does not destroy program currently in memory.

XX06   3   Both   A jump (JP) to a point in BASIC equivalent to issuing a SCRatch command. Does not re-size memory.

XX09   3   Both   A jump (JP) to a point in BASIC equivalent to the point reached via the jump at XX00 <u>except</u> that memory sizing and baud rate initialization do not take place. <u>On entry here, HL register pair should contain the address to be used as top of user program space, and the A register should contain the number of file channels to allocate.</u>

XX0C   3   CDOS   An illegal instruction (FF Hex, actually a RST 38H), which CDOS traps and uses to display an error message.

                SAB   A jump (JP) to the "IL" (Illegal command) error message.

                Both   If a CROMEMCO monitor resides in PROM at E000 hex, these 3 bytes should be changed to a jump (JP) to the breakpoint handler in the monitor. In case of failure in BASIC, the breakpoint information thus displayed might prove extremely valuable.

XX0F   3   CDOS   A jump (JP) to the CDOS warm-start at address 0000. Used by the BYE command to return to CDOS.

                SAB   A jump (JP) to location E008 Hex, the warmstart point in the CROMEMCO monitor. Used by the BYE command to exit to the

			monitor. CAUTION: BYE should not be used in systems without a monitor.
XX12	2	Both	The address pointer used by BASIC to locate the beginning of the Device Driver LIST. The user may use this address to find the DDLIST or may change it to force BASIC to use an alternate DDLIST.
XX14	2	CDOS	The address pointer to the beginning of the ERROR MESSAGE. The format of error messages in memory is as follows:

3 Bytes -- Call to error
            printer
1 Byte -- Error number
n Bytes -- ASCII message,
            terminated by a
            00 byte.

| XX14 | 2 | SAB | The address pointer to the beginning of the ERROR MESSAGES. The format of error messages in memory is as follows: |

3 Bytes -- Call to error
            printer
2 Bytes -- The error code,
            second byte
            first.

XX16	1	Both	The maximum number of channels which can be allocated at one time. Each I/O channel occupies 192 bytes of memory.
XX17	1	Both	The number of characters per line. The default value is 80. See SET instruction.
XX18	1	Both	The number of characters per TAB position. The

			default value is 20. See the SET instruction.
XX19	1	Both	The character which is used as a rubout, default value is 5FH (underline).
XX1A	1	Both	Default type of numeric variables and constants where 1=Integer, 2=Short Floating Point, and 4=Long Floating Point. The default value is 4.
XX1B	1	Both	Line delete character. The default value is control-U.
XX1C	1	Both	The carriage return delay (number of nul characters sent after a carriage return). The default value is 0.
XX1D	4	Both	Backspace echo - must terminate with a 0. The default value is back-space, space, backspace, 0.

DETERMINING THE ADDRESS OF A STRING

The address of any string may be determined as follows:

```
100 L=LEN(X$) : A=SYS(2)
or
200 A=SYS(2+0*LEN(X$))
```

When the length of a string is taken, its address is automatically placed in SYS(2). CAUTION: SYS(2) is also used to store the most recent I/O character (to console or file). So don't mix usages of the address capability with I/O statements, including TRACE. It is probably best to put the two statements on a single line or statement, as shown.

NOTE: In the unlikely event that a program is using SYS(2) in its old sense of "last character", only the least significant byte of SYS(2) receives the character. The most significant byte may be reset via SET 2,0, or masked off via BINAND (SYS(2),255).

Another Caution: Adding (or ENTERing) new statements or variables can cause string addresses to change. Use SYS(2) just before you need the address (typically for a USR call). You could even do the following:

```
A=USR(%430%,LEN(A$),SYS(2))
```

to pass the address and length of a string.

Also, A=LEN(A$(B)) : S=SYS(2) produces the address of the Bth character of the string A$.

ADDING DEVICE DRIVERS TO BASIC

Declaring Device Drivers

All I/O devices to be accessed from BASIC must be
declared in the device driver list. This list
starts at location DDLIST and each entry is 8 bytes
long. These entries are of the following form:

```
                 ┌─────────────────────────────────────┐
                 │          device name (ASCII)         │
        byte 0   │   first letter        second letter  │   byte 1
                 ├─────────────────────────────────────┤
                 │      base address of device letter    │
           2     │   LSB                          MSB    │   3
                 ├─────────────────────────────────────┤
                 │   may be used for device dependent    │
           4     │   information needed by driver         │   5
                 ├─────────────────────────────────────┤
           6     │               reserved                │   7
                 └─────────────────────────────────────┘
```

The device name (bytes 0 and 1) is used for finding
the correct device to OPEN. For example,
OPEN\2\"$LP" would instruct BASIC to search the
device driver list for device name "LP" (the $
indicates a device instead of a disk file). The
driver's starting address occupies the next two
bytes. Actual addresses and equated value may vary
depending on both revisions and which version of
BASIC is being used. The following two bytes are
reserved for use by the driver and their
function(s) are defined by the person programming
the driver. This device driver list can be placed
in PROM. These two bytes are thus not intended as
temporary storage areas.

A typical use of one of these bytes might be to
hold the actual device address to be accessed.
This would, for example, allow a general purpose
TU-ART driver to drive several ports each dedicated
to a different device.

The last two bytes of each table entry are reserved
for future use. The table itself is terminated by
the first nul driver name (Hex 00 in the first
character of the name).

Rules for Device Drivers

The actual device drivers must follow certain
prescribed rules. In particular, the first 16 byte
locations in the driver must contain address
pointers to the various routines (some of which are
required) in a BASIC driver. The 16 bytes define
the addresses of 8 different routines as shown in
the following table:

	(LSB)	(MSB)	
Byte 0	address of OPEN routine	Byte 1
2	address of CLOSE routine	3
4	address of SET STATUS routine	5
6	address of GET STATUS routine	7
8	address of PUTC routine	9
10	address of GETC routine	11
12	reserved (should be 0)	13
14	reserved (should be 0)	15

If any of the routines noted above are not defined
for a given driver, the corresponding address field
should be set to 0000H. However, if a routine is
(or must be, see below) defined but does not do
anything, the corresponding address must contain
the address of an XOR A/RET instruction sequence.

A description of what must be accomplished by the
various routines follows. Parameters which may (or
must) be passed between BASIC and the driver
routines are always passed in registers, or in the
Extended File Control Block (EFCB).

Most of the information needed by the driver
subroutines may be obtained from (or saved in) the

Extended File Control Block (EFCB).

Register usage is covered in the description of each command routine below:

OPEN: This routine should perform any processing required to initialize the device. For example, a line printer driver would typically issue a form feed on open.

On Entry: (A') = no. of parameters (Ø,1,or 2)
(IY) = ADDR of EFCB
Ø, 1, or 2 parameters are passed to the routine in locations EFCBP1 and EFCBP2 of the EFCB.

On Return: A not equal to Ø says can't open. This is a user-trappable run time error.

CLOSE: This routine should perform processing necessary to "shut down" the device. For example, a paper tape punch driver might punch out a trailer piece of tape.

On Entry: (IY) = ADDR of EFCB

On Return: A not equal to Ø says can't close. This is a user-trappable run time error.

PUTC:
GETC: These routines perform byte-by-byte transfers to and from the device. Devices requiring a buffer may use the buffer (and/or extended buffer) in the EFCB.

On Entry: (IY) = ADDR of EFCB
(A) is character to be written (PUTC only)

On Return: (A) used for character read (GETC only)

```
          SSTAT:   (SPOS)
                   This routine is used to set the status
                   of the device.  For example, a DAZZLER
                   driver might use this to set the X/Y
                   screen position.

                 . On Entry:    (A') = no.  of parameters
                                (Ø,1,or 2)
                                (IY) = ADDR of EFCB
                                Ø, 1, or 2 parameters are
                                passed to the routine in
                                locations EFCBP3 and
                                EFCBP4 of the EFCB.

                   On Return:   A not equal to Ø says
                                invalid status request.
                                This is a user-trappable
                                run time error.

          GSTAT:   (GPOS)

                   On Entry:    (IY) = ADDR of EFCB
                                (A') = which status is
                                requested

                   On Exit:     (HL) contains appropriate
                                status value.  A not equal
                                to Ø sets a user-trappable
                                runtime error.

          NOTE:  Registers IX and IY cannot be changed by any
          of these routines.

          The EFCB referred to has the following format:

          EFFCB   :   DS 1     ;Ø=not in use, 1=in use
          EFCBDA  :   DS 2     ;Device Driver Address,
                               ;bytes 2&3 From DDLIST
          EFCBDD  :   DS 2     ;Device Dependent info,
                               ;bytes 4&5 from DDLIST
          EFCBP1  :   DS 2     ;\These are the two optional
                               ;parameters
          EFCBP2  :   DS 2     ;/P1 & P2 as used in the
                               ;"OPEN" statement
          EFCBP3  :   DS 2     ;\These are the 2 optional
                               ;parameters P1 & P2
          EFCBP4  :   DS 2     ;/used in PUT, GET, PRINT,
                               ;and INPUT statements
          BUFFER  :   DS 179   ;Available to user for
                               ;accumulating individually
                               ;passed bytes into a buffer
```

When the proper DDLIST entries have been made as shown, operation of the driver is as follows:

OPEN\1,A,B,\"$DR"
Calls the OPEN routine of driver "DR".
EFCBP1 = A, EFCBP2 = B
A' register = 2, IY= ADDR of EFCB
If OPEN is not needed, it must consist of an XOR A, RET sequence.

PRINT\1,X,Y\"HELLO"
The Set Status routine of "DR" is called with EFCBP3 = X, EFCBP4 = Y (X and Y are converted to integers first).
A' = 2, IY = ADDR of EFCB
An XOR A, RET sequence should be executed at the completion of Set Status.
Next, the PUTC routine of "DR" is called 7 times, once for each character to be transmitted plus once with a Carriage Return and once with a line feed.
A = character, IY = ADDR of EFCB

INPUT\1\A$
Set Status routine of "DR" is called with EFCBP3 = %FFFF%,
EFCBP4 = %FFFF%,
A' = 0, IY = ADDR of EFCB
An XOR A, RET sequence should be executed.
Next, the GETC routine of "DR" is called repeatedly, expecting one byte to be returned in A. This continues until a terminator (CR,LF,FF,NULL) is transmitted, or until more than 132 characters have been transmitted.

A = IOSTAT(1,n)
where 0≤n≤255
The Get Status routine of "DR" is called. A' contains n, the requested status parameter and IY = ADDR of EFCB. The status value to be returned should be placed in HL.
An XOR A, RET sequence should be executed last.

CLOSE\1\
The CLOSE routine of "DR" is called. If CLOSE is not needed, an XOR A, RET sequence must be provided.

Note that if PUT and GET were used in the above
calls instead of PRINT and INPUT, binary bytes
would be transmitted according to variable type and
no carriage control information would be sent.
I.E., PUT\1\V would transmit two bytes if V is an
integer, four bytes if V is short floating point,
and eight bytes if V is long floating point. No
carriage return or line feed is sent. Also note
that PRINT\1\"A","B" will not transmit the comma
but will send the proper number of spaces to place
"B" in the next tab field.

DDLIST (DEVICE DRIVER LIST)

ADDR	CODE				
		;	NOTE:	THE ADDRESSES SHOWN IN THIS LISTING	
		;	MAY VARY FROM VERSION TO VERSION, but the		
		;	format is identical. The actual starting		
		;	address may be found in locations 0412,		
		;	0413 for CDOS BASIC and 8012, 8013 for		
		;	Stand Alone BASIC		
		DDLIST:			
4B6C	53	DDCNSL	DB	'SY'	;'SYSTEM' = CONSOLE
4B6D	59				
4B6E	E84A		DW	DRTUART	
4B70	00		DB	0,0	
4B71	00				
4B72	00		DB	0,0	
4B73	00				
4B74	54		DB	'T5'	;2ND TUART PORT
4B75	35				
4B76	E84A		DW	DRTUART	
4B78	50		DB	50H,50H	;BOTH BYTES GET ADDR ;OF 2ND TUART
4B79	50				
4B7A	0000		DW	0	;RESERVED BY SYSTEM
		;			
		; CDOS SYSTEM PUNCH READER, AND LIST DRIVERS			
4B7C	50		DB	'PU'	
4B7D	55				
4B7E	9C4A		DW	DRCDPU	;CDOS PUNCH DRIVER
4B80	0000		DW	0	
4B82	0000		DW	0	
4B84	52		DB	'RD'	
4B85	44				
4B86	B54A		DW	DRCDRD	;CDOS READER
4B88	0000		DW	0	
4B8A	0000		DW	0	
4B8C	4C		DB	'LP'	
4B8D	50				
4B8E	CA4A		DW	DRCDLP	;CDOS LIST DRIVER
4B90	0000		DW	0	
4B92	0000		DW	0	
4B94	00	DDEND:	DB	0	;END OF DDLIST-NOTE: IF DDLIST IS EXTENDED, THIS NULL should be overwritten and then placed at
		;			
		;			
		;			
		;			

```
                        ;                                      end of the new DDLIST
          4B95                      DEFS   64                  ;RESERVE SPACE FOR
                                                               ;MORE DRIVERS
          FFFF                      IF     SOSVER              ;--------------------

                                    DW     !DUMMY
                                    ELSE                       ;--------------------
```

CDOS PUNCH, READER, AND LIST DRIVER INTERFACES

```
ADDR      CODE       ; SAMPLE DEVICE DRIVERS
                     ;
                     ; PUNCH
                     ;
                     DRCDPU:
4A9C      E64A                      DW     $DUMMY
4A9E      E64A                      DW     $DUMMY
4AA0      E64A                      DW     $DUMMY
4AA2      E64A                      DW     $DUMMY
4AA4      AC4A                      DW     $PUPC      ;PUNCH  OUT  CHARACTER
4AA6      0000                      DW     0
4AA8      0000                      DW     0
4AAA      0000                      DW     0
                     ;
4AAC      5F         $PUPC:         LD     E,A        ;GET CHAR TO E REG AS
                                                      ;CDOS EXPECTS
4AAD      0E04                      LD     C,4        ;CDOS PUNCH ENTRY
                     $PULPJ:
4AAF      F5                        PUSH   AF
4AB0      CD0500                    CALL   CDOS
4AB3      F1                        POP    AF
4ABF      C9                        RET
                     ;
                     ;
                     DRCDRD:
4AB5      E64A                      DW     $DUMMY
4AB7      E64A                      DW     $DUMMY
4AB9      E64A                      DW     $DUMMY
4ABB      E64A                      DW     $DUMMY
4ABD      0000                      DW     0
4ABF      054A                      DW     $RDGC      ;RDR GET CHAR ROUTINE
4AC1      0000                      DW     0
4AC3      0000                      DW     0
                     ;
                     $RDGC:
4AC5      0E03                      LD     C,3        ;CDOS READER GETC
                                                      ;ENTRY PARM
4AC7      C30500                    JP     CDOS       ;READY  TO  GO... CHAR
                                                      ;RTND IN A
```

```
                        DRCDLP:
4ACA    DF4A                    DW      $LPOP       ;OPEN ROUTINE
4ACC    E64A                    DW      $DUMMY
4ACE    E64A                    DW      $DUMMY
4AD0    E64A                    DW      $DUMMY
4AD2    DA4A                    DW      $LPPC       ;AND LP PUTC ROUTINE
4AD4    0000                    DW      0
4AD6    0000                    DW      0
4AD8    0000                    DW      0
                        $LPPC:
4ADA    54                      LD      E,A         ;MOVE CHAR TO E
4ADB    0E05                    LD      C,5         ;CDOS LIST WRITE
                                                    ;ENTRY PARM
4ADD    18D0                    JR      $PULPJ
                        $LPOP
4ADF    3E0C                    LD      A,ASCFF     ;ISSUE A FORM FEED ON
                                                    ;OPENING FILE
4AE1    CDDA4A                  CALL    $LPPC
4AE4    AF                      XOR     A
4AE5    C9                      RET                 ;RET WITH 0 STATUS
                        $DUMMY:
4AE6    AF                      XOR     A
4AE7    C9                      RET
```

<u>ACCESSORY I/O DRIVERS</u>

The DDLIST contains five drivers which may be
accessed by the user. BASIC treats these drivers
as files. They must therefore be OPENed and
CLOSEd, but may not be CREATEd (because they
already exist). Note that the first three drivers
are not available in the Stand Alone version of
BASIC.

<u>Line Printer Driver</u>

The system line printer can be accessed by OPENing
a file named $LP. This driver uses the CDOS system
call to output data to the line printer. INPUT and
GET instructions must not be used with this file.

<u>Punch Driver</u>

The system punch may be accessed by OPENing a file
named $PU. This driver uses the CDOS system call
to output data to the punch. INPUT and GET
instructions must not be used with this file.

<u>Reader Driver</u>

The system reader may be accessed by OPENing a file
named $RD. This driver uses the CDOS system call
to input data from the reader. PRINT and PUT
instructions must not be used with this file.

<u>Console Driver</u>

The system console may be accessed by OPENing a
file named $SY. This driver <u>bypasses</u> CDOS and
allows the user to input and output data from/to

the console directly by using port Ø (UART status)
and port 1 (UART data).

TU-ART Driver

There is a general purpose TU-ART serial I/O port
driver in the DDLIST called T5. It assumes a TU-
ART addressed at port 50H and may be used for any
serial device provided the baud rate is initialized
prior to its use with an OUT instruction.

Example:

        10 OUT %50%,4 : REM SET TU-ART TO 300 BAUD

        20 OPEN\2\"$T5" : REM SETS UP DEVICE #2 FOR SERIAL I/0

The driver for the system console at port Ø is
labeled "$SY". To switch between the console and a
second terminal:

        20 OPEN\1\"$SY"

        30 OPEN\2\"$T5"

This will allow you to later say PRINT\A\"MESSAGE",
where A can be changed via software to be 1 for the
console and 2 for another terminal.

CHANGING THE NUMBER OF I/O CHANNELS

BASIC carries 4 I/O channels in addition to the console. One I/O channel is needed for each file which is OPENed at the same time. Each I/O channel occupies 192 bytes of memory.

If you wish to change the number of channels in use, type the following commands while in BASIC:

Disk System	Stand Alone System
POKE%416%,X	POKE%8016%,X
A=USR(%400%,0)	A=USR(%8000%,0)

where X is the number of I/O channels desired in addition to the console. The POKE command puts the number of channels desired into the proper location and the USR function reinitializes BASIC.

On a disk system, you can now save this new version of BASIC by returning to CDOS and typing:

>>BYE

A.SAVE BASIC1.COM 76

LOADING 16K BASIC FROM PROM

I. Memory Requirements:

    1. 16K PROM starting at location 8000H.
    2. At least 8K RAM starting at location 0.

II. System Set Up:

    A. CROMEMCO 8K BYTESAVER II PROM board:

        1.   Turn PROM program power switch 'OFF'.

        2.   Turn program enable switches 'OFF'.

        3.   Plug the 16 CROMEMCO 16K BASIC PROMs into the boards in numerical order.

            board one –
            8000 into PROM socket 0
            8400 into PROM socket 1
            etc.

            board two –
            A000 into PROM socket 0
            A400 into PROM socket 1
            etc.

        4.   Set LOGICAL ADDRESS BLOCK SELECT to 8000H & A000H

            BYTESAVER II switches:

            board one – 15=1, 14=0, 13=0
            board two – 15=1, 14=0, 13=1

    B. 16KPR board:

        1.   Set LOGICAL ADDRESS BLOCK SELECT to 8000H.

            switch 15=1
            switch 14=0

2.   Plug the 16 CROMEMCO 16K BASIC PROMs
      into the board in numerical order.

      8000 into PROM socket 0
      8400 into PROM socket 1
      etc.

III. All systems:

1.   Set the jump start address on the ZPU
      board to 8000H.

      Switch A15=1
             A14=0
             A13=0
             A12=0

2.   Select bank 0 on the PROM board (switch 8
      on).

IV. Method:

1.   Depress the return key on the console
      several times so that CROMEMCO 16K BASIC
      can determine the baud rate of your
      terminal.

2.   CROMEMCO 16K BASIC will respond with a
      prompt (>>) to indicate that it is ready
      to accept a command.

3.   Refer to the section on Getting Started
      for information on how to proceed from
      this point.

I N D E X